UNIVERSITY OF OSLO
Department of informatics

User Interaction Middleware Architecture and Protocol

Cand Scient thesis

Jonas Lindholm

1. May 2007

Acknowledgments

I would like to thank my supervisor Øystein Haugen for encouraging and valuable advices through our discussions. You have guided me in the right direction.

I also would like to thank my wife and my family for being patient while I have been working with my thesis.

Abstract

This thesis proposes an architecture for user interaction. The architecture is a message-based middleware that controls and manages the interface. The architecture has built-in features like single responsibility, state management and asynchronous messaging. We show that this eases the development of reliable and dynamic user interaction applications in a thread ignorant environment. The controlling of the interface is performed by UML state machines. The architecture is shown to be modular and extendable. New parts of the application, both behaviour and graphical elements, can be added at runtime.

A case application has been re-implemented using the architecture of the thesis, and the re-implementation is compared with the analysis of the original.

The thesis covers the technical part of user interaction rather than the graphical. The problem addressed is the handling of events and distribution of actions to perform state management in interaction applications.

Table of contents

List of figures

List of Tables

1. Introduction

Motivation

The motivation for this thesis is to get a better method to control and execute user interaction. There is a need for better control both in designing and developing of user interaction applications. State management is simple with smaller applications but can be hard in medium to complex solutions.

The focus of this thesis is the area between the graphical interface, with the look and feel, and the functional core, which implements domain dependent concepts. The middleware developed does not manage the layout and rendering of elements on the screen.

Goal

The goal is to develop an architecture and a method of how to develop user interaction which is efficient, easy to control, easy to use for developers and easy to extend.

Mode of procedure

In the work with the thesis I have been performing experiments to investigate properties of different architectures for use in user interaction. The architecture proposed here has been evolving through the work of the thesis. The starting point was that controlling user interaction by state machines should be powerful. When a user interacts with an application the state of the application changes and has to be managed in some way. State machines should be a useful solution. The question was how this could be achieved and which structure was most effective.

The investigation started with simple example applications for exploring and developing the architecture. A real case application were analysed to find areas where user interaction could be problematic in a common used architecture. Parts of the analysed application were re-implemented, when the architecture of the thesis were more mature, to show how the problematic areas could be solved using the architecture developed. The architecture has been changing with every new experiment which was implemented and the experience gained has transformed the structure to the result which is presented here.

Structure of the thesis

Background to user interaction is discussed before the analysis of the case and the description of the proposed architecture.

Chapter 2 gives a background to the development of user interaction and related work in the area of use of state machines in user interaction. Patterns used in user interaction, graphical user interface libraries and architectures are discussed.

In chapter 3 there is a case of a Case Management application. It is an analysis of a real and full scale application with a discussion of the problems found in the area of user interaction in the application. The architecture and the user interaction mechanism are described.

Chapter 4 presents the architecture this thesis proposes. It is a description of the structure of the architecture and of the behaviour the architecture has. Implementation and principles for using the architecture is discussed.

The re-implementation of the analysed application is described in chapter 5. Selected parts of the analysed application is re-implemented and described. There is also a comparison of the approaches.

Conclusion and further work is found in chapter 6.

2. Background

The behaviour is the most important asset to both users and developers of applications.

“A computer is an organization of elementary functional components in which, to a high approximation, only the function performed by those components is relevant to the behavior of the whole system.” [Simon '96]

Herbert Simon observes that behaviour, not the internal structure, is the most important quality of any system. From a user’s perspective, behaviour is the most important part of the application. The behaviour has to be consistent to the user; different user patterns in a task must lead to the same result.

In a development perspective behaviour is the specification of the interaction, the sequence each individual operation occurs in and the possible states every part can have. The developer needs to be able to express the behaviour of the application in a way that is consistent to the user and verify that the implementation follows the specification.

In the following there is an introduction to the user interaction field. There is also a background to the mechanism used when developing user interaction applications. We will look into the field of human-computer interaction and why behaviour is important to both users and developers.

2.1 Development in Human Computer Interaction

Applications and the interaction with computers are changing. Computers are found in small devices, mobile telephones are gaining more processing power to perform more advanced tasks. Interaction over networks is common in most applications developed today. Users demand to be available and in contact with the internet most of the time. Not only when interacting with the workstation computer, but also during travelling and time off work. The border of where one application starts and another stop is becoming blurring. This applies to not only to software but also to the computers or devices itself.

2.1.1 Blur of applications

Applications are moving towards more dynamic and flexible structures where they can be extended to meet the changing needs of the user. Applications need to be changed at runtime, when the user needs additional functionality and not in releases in regular intervals.

The new dynamic applications define extension points where the extended behaviour is plugged in on user request. Examples of this functionality are Eclipse and Firefox web browser [Firefox] where plug-ins can be added to extend the functionality. The developers at Eclipse have a set of rules for design and the first rule is about contribution: “Every thing is a

contribution" [Gamma and Beck '04]. The Eclipse runtime is a tiny kernel where contributions are added as plug-ins. The plug-ins offer in turn extensions where other plug-ins can contribute.

This is a change of the idea of an application where applications are self-contained units. The limits of applications can becomes blurred as new features are added not initially present.

2.1.2 Blur of products

More and more electronic products for the consumer market come with some kind of interaction mechanism for connection to a network or computer. Watches having sensors for heart rate monitoring is common in training centres. Other watches come with GPS units for positioning and distance calculation [Hyman '05]. It is not unthinkable that different electronic devices will communicate with each other and use the functionality of other devices in reach. The functionality of the devise will be changed due to the other devises in the surroundings.

Interaction with computers that we are used to today is changing and new specialised types of devises are appearing on the marked. Game consoles with more than one screen. Remote controls have motion sensing capabilities which allow users to interact with and manipulate items on screen via movement and pointing [NintendoWii].

2.1.3 Human to human interaction

The complexity of user interaction increases in a world with accelerating number of interactions with computers and other devices each day.

"By 2010, 70 percent of the populations in developed nations will spend 10 times longer per day interacting with people in the electronic world than in the physical one." [Gartner '06]

The prediction asserts that more and more communications between humans will be carried out in the electronic world. Messages through e-mails, instant messaging, and applications not developed yet. The number of applications in concurrent use increases and more time is spent monitoring all channels for communication. The users are faced with more real time data than before.

Applications need to handle a rising number of messages simultaneous with the concurrent computation of the user. This changes the requirements for the architecture of the applications. Parallel computation and asynchronous messaging is needed to handle the rising complexity of interaction in applications.

2.2 Related work

State machines have been used for a long time in software engineering. David Harel's development of the statechart diagram in 1987 [Harel '87] added three important elements to

the conventional state-transition diagrams; hierarchy, concurrency and communication. This was needed to manage models of complex systems in a modular approach. State machines are in common use in the software industry but not always described as in the UML 2.0 specification. The idea about state machine for managing the control of the state and navigation is widely implemented in applications and framework. Microsoft research has implemented an engine for abstract state machines [Börger and Stärk '03] for use in the .NET environment [Gurevich '04] and IBM uses state machines in the WebSphere Process Server [WPS '07] for modelling business processes [Beers and Carey '06].

In the following we present some approaches where state machines are used in software engineering and user interaction.

2.2.1 JStateMachine

In a proceeding of ForUse 2003 David J. Anderson and Brían O'Byrne wrote about using Statecharts with Feature Driven Development (FDD) [Anderson '03]. FDD is an agile method approach used in combination with David Harel's state charts notation [Harel '87] to implement user interaction. An engine, JStateMachine [Anderson '07], was used to run user interface designs models. The engine was the runtime environment for the models.

The use of state chart modelling was described as a "clean, well-defined rigor to the definition of a user interface and the implementation in the JStateMachine engine brought reliable execution and enabled a table-driven, soft-coded approach for making late modifications to the design and implementation" [Anderson '03]. It seems they were satisfied with the approach but the reference to the web sites of the engine is outdated and seems not to be in active use.

In a whitepaper by David J. Anderson [Anderson '00] about using the Model-View-Controller architecture in web interactions, Anderson describes the characteristic and methods of using state charts for user interaction design and implementation. The example implementation is a web application architecture which uses a controller to handle the state of the application. The architecture forces the developer to model the application as one large state machine with all states the application holds. In whitepaper there are a reference a company named Statesoft (see below) and it appears that Statesoft's product ViewControl is the successor to the architecture described in the paper.

It is also mentioned that there have been an experiment with using Together [Borland '07] to generate an XML definition of the state representation with transitions and events.

2.2.2 Statesoft - ViewControl

ViewControl [Statesoft '07] is a plug-in for Microsoft Visual Studio that enables developers to model user interactions as a state machine with a visualization of the state machine. A prototype can be generated from the design (state machine) to evaluate the usability of the user interaction. There are both a .NET and a Java version of the runtime. Statesoft claims

that the develop-test-deploy cycles are 2-3 weeks and that the development cost for the presentation is reduced by average 50%.

The ViewControl tool is simply a state machine editor for navigation. The tool generates a navigation controller component that holds references to the JSP/.NET pages that are going to be shown in the transitions between the states. The tool has support for modelling sub states.

2.2.3 Spring Web Flow

Spring Web Flow (SWF) is a component in the Springframework [Springframework '07] [Johnson and Hoeller '04] for managing definition and execution of flow in a web application. Spring web flow is an engine which can be integrated with a number of web frameworks as Struts, Spring MVC, Java Server Faces etc. The flow is captured in a self-contained module that can be reused in different situations.

The Spring Web Flow has predefined state types which are used to define the flow, End state, View state, Action state etc. The state of the flow is stored in the HTTP session, a database or a client side form field between the executions of the flow. Alternative flows can be defined as sub flow. A sub flow is a self-contained module and is a kind of substate machine or reusable composite state which is used in the overall application flow. There is no functionality to define a state machine which is executed in isolation.

2.2.4 JavaFrame

JavaFrame is a framework for modelling and running systems using state machines [Haugen and Møller-Pedersen '00]. It consists of classes for supporting modelling and for the runtime system. JavaFrame were developed as a research project within Ericsson. Additional framework has been developed on top on JavaFrame. ActorFrame [Husa and Melby '04] is a framework for execution of services, developed from domain specific concepts, in a service oriented network. ServiceFrame is an extension to use the framework in a J2EE environment with a proposed UML profile for mapping of the ActorFramework concept to J2EE [Melby '03].

A code generator has been implemented for transformations of UML to Java for the JavaFrame runtime [Willersrud '06] based on a UML profile for JavaFrame. The generator was implemented as a plug-in for Rational Software Modeler based on Eclipse technology.

2.3 Common Patterns in User Interaction

In software engineering design patterns are used to describe repeatable solutions to commonly problems in system development. Patterns originated as an architectural concept as an aid to design cities and buildings by Christopher Alexander. “Each pattern describes a

problem which occurs over and over again in our environment, and describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same way twice" [Alexander et al. '77].

2.3.1 Design Patterns

The design patterns in computer science are documented in a commonly used form. They have a named description of a problem, a solution, when to apply the solution, and how to apply the solution in new contexts. This form origins from the first book about design patterns referred to as the Gang of Four (GoF) [Gamma '94] but other pattern form is in common use [Fowler '03].

Design Patterns used in the thesis:

- **Observer** – Define a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
- **Mediator** – Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Composite** – Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Adapter** – Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.
- **Chain of Responsibility** – Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

2.3.2 Model-View-Controller

In software engineering user interaction has followed common separation known as Model-View-Controller (MVC) first described by Trygve Reenskaug [Reenskaug '79a] [Reenskaug '79b]. User interaction development has since that time connected the different user interaction solutions to this separation. New proposals for solutions have been related to the MVC architecture pattern.

MVC as described by Reenskaug explains model as an object "or it could be some structure of objects". The model represents knowledge, the domain of the application. View is a visualisation of the model, acting as a "presentation filter". "A controller is the link between a user and the system." The information is presented by letting the "relevant views to present themselves in appropriate places on the screen". "The controller receives such user output,

translates it into the appropriate messages and pass these messages on to one or more of the views."

In the relationship described above, the view "is attached to its model" and gets the data from the model by "asking questions". "It may also update the model by sending appropriate messages. All these questions and messages have to be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents." This means that the view has knowledge about the model but the model is independent of the view. As illustrated in the citations above, the controller should handle the user input and delegate the information to the view.

2.3.3 MVC variations

Different architectures allow the three components in MVC to interact in different ways. Interaction and responsibility varies in different MVC implementations.

Sun Blueprint

Figure 2-1 below shows a common implementation as shown in the Sun Blueprint Catalog [Singh et al. '02].

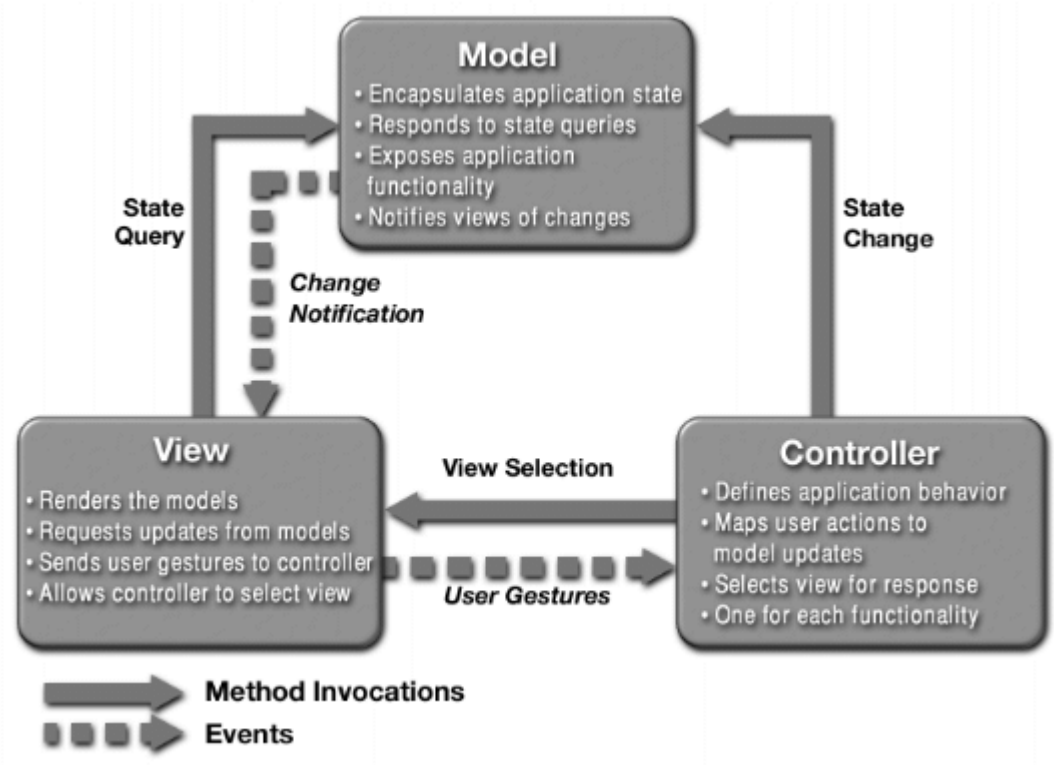

Figure 2-1: The Model-View-Controller architecture from the Sun Blueprint Catalog[1]

[1] Figure 2-1 is adapted from http://java.sun.com/blueprints

Figure 2-1 shows the separation of responsibilities among the components in the MVC architecture proposed by Sun. The logic for the different responsibilities is separated from each other. This eases the maintainability. The view forwards the user events to the controller and uses the model for queries. A view could also be notified if there is a change in the model. The controller updates the model and invokes the view. The model holds the application state and must be able to inform of changes to interested views. The view could either use the query or the notification mechanism, called “pull model” or “push model”, or a combination of both.

Here is a scenario for an interaction in the Java SE platform. A view is registered as a listener to the model. The controller is bound to the view and holds a reference to the model. This happens when a user interacts with view:

- The view calls the controller.
- The controller updates the model due to the changes in the view.
- If the model has been altered it notifies the interested views.

Cocoa application environment

Another variant of a MVC architecture is to place the controller between the model and the view as in the Cocoa application environment from Apple [Cocoa '07]. The argument for this is that it is easier to reuse model and view objects in an application if they are completely separated.

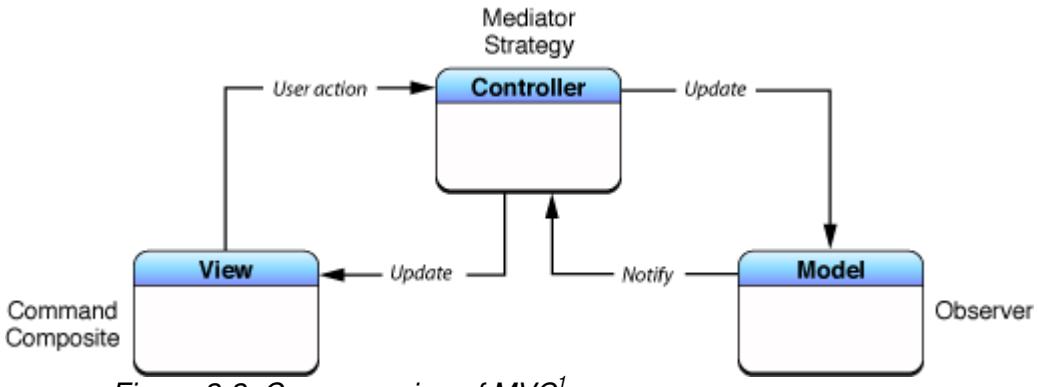

Figure 2-2: Cocoa version of MVC[1]

In Figure 2-2 the Cocoa version the controller mediates the flow between the model and the view in both directions.

Presentation-abstraction-control architecture

The presentation-abstraction-control (PAC) model [Coutaz '87] has a structure for user interaction systems in the form of a recursive hierarchy of cooperating agents. Each agent is responsible for a specific aspect of the applications functionality and consists of three

[1] Figure 2-2 is adapted from the Cocoa documentation: http://developer.apple.com/documentation

components: presentation, abstraction, and control. The functional core is implemented by the abstraction component. The presentation component interacts with the GUI library. The control manages the relationship between the PAC components and the controller may itself be a PAC hierarchy. The PAC concept favours a modular decomposition of the user interface and provides more directions than the MVC of how to form the composition of the user interactive system. There are no known libraries or components based on PAC and it does not describe a particular control and data propagation mechanism [Markopoulos '97].

2.3.4 Web applications and MVC

Web application MVC architectures is implemented using two different models [Cavaness '02] with different maturity of the architecture. Model 1 in Figure 2-3 below has a decentralized controller and navigation logic. Each view is responsible for handling user requests and the navigation. This makes Model 1 application less modular, cohesive and difficult to maintain. The Model 2 centralizes the controller and navigation logic. Front Controller and a Application Controller are often used [Alur et al. '03] where the Front Controller is the entry point for requests and the Application Controller handles the action and view management. The Application Controller delegates the request to the appropriate worker for the specific task for execution. After the worker has finished, the controller dispatches to the view for rendering using the model.

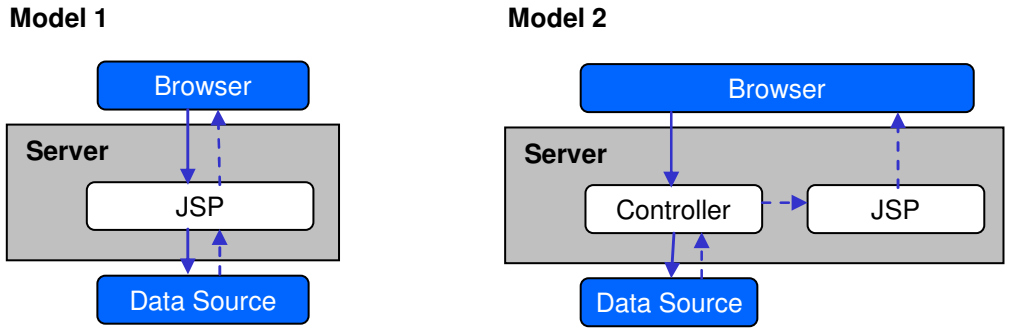

Figure 2-3: Model 1 and Model 2 of web applications

The Model 2 is closer to the original ideas about the MVC architecture in ownership and responsibility. It has a central controller that handles the user request and delegates to the view. But the view has no reference to the model and can not be informed of any changes. This makes the Model 2 more like the Cocoa version of MVC.

2.3.5 MVC and pattern definition

Many have experienced that the MVC can be implemented with different relationship and usage of the separation of the components in the MVC [Eckstein '07] (The Cocoa environment, Model 1 and Model 2 as previous mentioned). Reenskaug suggested a MVC Pattern Language to cover the different approaches [Reenskaug '03].

The Model 2 of MVC, used in a web application context, is implemented by using the J2EE patterns Front Controller and Application Controller (see 2.3.6 - J2EE core patterns below). Many GUI libraries use the Observer pattern and the Mediator pattern among others.

In comparison with the Gang of Four design patterns MVC consists of a number of patterns in the different MVC implementations. MVC is stated to be an architecture pattern which is larger in scope than the design patterns [Avgeriou and Zdun '05].

The MVC implementation used in the Sun Blueprint Catalog, Figure 2-1, makes the parts in the MVC architecture more dependent of each other than the in the Cocoa solution, Figure 2-2, and the Model 2, Figure 2-3, solution. The view, which registers for notification of model changes, is bound to the model. But in the Cocoa solution the view is just bound to the model by the controller.

2.3.6 J2EE core patterns

Sun has captured and described the common patterns used for implementation of applications using Java/J2EE [Alur et al. '03]. These patterns are the foundation for many frameworks and applications in the Java sphere [Fowler '03] [Johnson '02]. They are also used in the reference application describe in chapter 3 - Analysis of a Case Management application.

Figure 2-4 below shows an overview over the major patterns used in a presentation layer. (Other J2EE patterns exists but not described here) The name of the boxes is name of the patterns in Sun's J2EE Core Patterns except for View that is included for completion of the description.

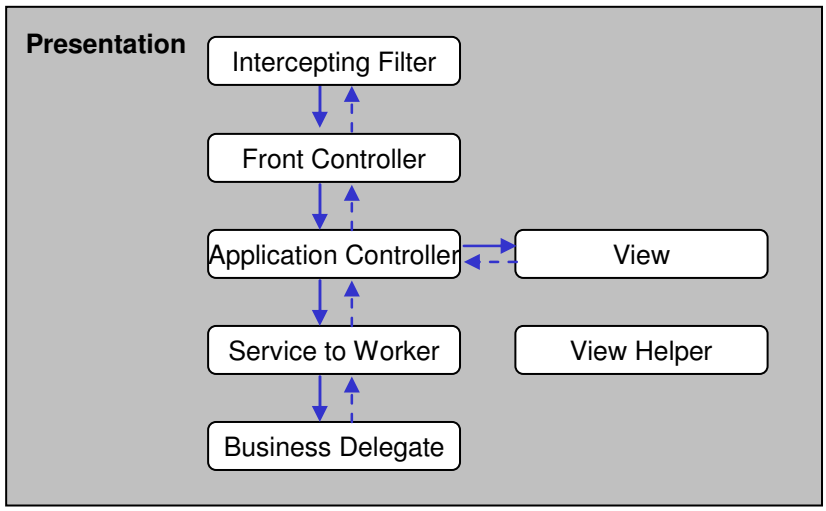

Figure 2-4: Presentation architecture

The architecture components used to handle the presentation logic:

Intercepting filter – Pluggable filters process the incoming request from the user. Intercept the request and add pre and post processing without requiring changes to core request

processing code. Performs security checks etc. Once validated, the request is passed on to the controller for actual processing.

Front Controller – Used as a controller to handles the requests when they have passed the filters. Performs validation of user input etc. Selects an appropriate application controller and delegates the request to it.

Application Controller – Invokes the Action that matches the request. The controller holds a reference to all actions in the application.

Service to Worker – The worker the application controller chooses to use to fulfil the request. The worker implements the presentation logic is the connection to the business layer.

View – Process the formatting of the response.

View Helper – Data container that passes information between the Service to Worker and the View. The helper is the views intermediate model and serves as business data adapters.

Business Delegate – handles the interaction with the business layer. It hides the underlying implementation details of the business services.

2.4 Graphical user interface

Graphical use interface (GUI) allows for graphical interaction with computers using images, widgets and text etc. to represent information and the actions available to the user. There are several GUI libraries, which controls the graphical interaction, in common use today. Some of the libraries are cross platform in regard of that they could run on a number of different platforms. The cross platform ability works in different ways for the different libraries; from a compile time platform dependency to a requirement of a runtime on the specific platform.

Definitions

- Widget - is a graphical interface element that a user interacts with, a button, text box, window etc.

2.4.1 User interface libraries

On the Java platform the Java Foundation Classes [Walrath '04], mostly refer to as Swing, is the common GUI library which comes with the Java API. This has been the most used library on the Java platform but SWT, described below, has gained popularity lately with the use of the Eclipse platform.

In Swing and SWT event handling in user interaction is accomplished by using a call-back mechanism described as the observer pattern (see 2.3 - Common Patterns in User

Interaction). Objects register for notification to user events on GUI widgets. When a user triggers an event on a widget the registered objects is notified.

Other popular GUI libraries use more or less the same mechanism. The C# language on the Microsoft .NET platform uses a “delegate” for event handling [Sells and Griffiths '05]. A delegate is a kind of type safe function pointers which wraps a method and passes it to another object.

Four steps are involved when using delegates:

1. Declare a delegate with the same signature as the method that’s going to be encapsulated.
2. Define the method that will be used.
3. Create the delegate object and plug in the method to be encapsulated.
4. Call the encapsulated methods through the delegate object.

An event handler in C# is a delegate with a special signature.

The .NET framework also has a XML based language for defining the user interface (XAML) [Petzold '06]. But this markup language does not change the event mechanism used in the .NET framework.

Another popular GUI library is the Qt cross platform C++ library[1] [Trolltech '07]. It consists of class libraries as well as a GUI builder. Qt does not use the same call-back functionality as common in the Java and the .NET environments. Qt uses a signal and slot mechanism for handling interactions events. A signal is emitted when an event occurs. A slot is a function that is called in response to a particular signal. The signature of the signal must match the signature of the slot. Signals and slots are loosely coupled: A class which emits a signal neither knows nor cares which slots receive the signal. A signal and a slot are connected by registering them using a registration method (connect) or using the auto-connection facilities which is based on naming conventions.

2.4.2 Standard Widget Toolkit

The Eclipse environment uses a GUI library called Standard Widget Toolkit (SWT) for displaying user interface widgets. It is a thin compatibility layer which interacts with the underneath platform. SWT uses the native controls of the user interface and provides a common API for all the platforms that are available. SWT provides the foundation of the entire Eclipse UI.

SWT uses an event notification pattern for GUI components to communicate with the application and other GUI components. It is an implementation of the Observer pattern where you express interest in a particular event by registering a listener with a widget. A listener is an observer that is notified when an event occurs. Different listeners define

[1] A Java binding for the Qt library, Qt Jambi, is under development.

interfaces for handling different events. An adapter implements the listener interface and performs actions of choice.

SWT provides 16 different events. It ranges from the usual KeyEvent – keys being pressed and released on the keyboard to TreeEvent which are sent as result of trees being expanded or collapsed and HelpEvent which occurs when help being requested for a widget.

SWT, as the other GUI libraries, uses one thread for communication with the GUI, the Event Dispatch Thread. This thread is responsible for rendering graphic on the screen. If another thread is used to access a GUI component an exception is thrown. Tasks run on the event dispatch thread must finish quickly otherwise unhandled events back up and the user interface becomes unresponsive. Heavy processing should be executed in another thread. But data processing is performed in the Event Dispatch Thread by developers for different reasons; lack of experience, laziness etc. But the important thing is that the developer has to be aware of the problem, and has to follow the guidelines in how to execute heavy processes in the user interaction frameworks. This is a fallible approach which might cause problems and lead to less responsible applications.

2.5 User Interaction clients

Architectures of user interaction applications are divided in where the computation is executed; applications which runs on the client computer and applications which mainly rely upon server execution. Applications that run on the client computer and just have connections to backend systems are called thick clients. Web applications and mainframe terminal applications are the opposite where the computation is execute mainly on a server. The client side computation is mainly about rendering of the graphical elements.

Thick clients (also known as “fat client” or “rich client”) are a client that computes the most of the processing by itself and relies on the server mostly for storage of information.

The understanding of a “rich client” has recently changed to a hybrid between a thin and a thick client. With the introduction of AJAX (Asynchronous JavaScript and XML) and similar technologies for more frequent interaction between the client and the server more of the processing moves to the client side. The user experiences quicker response from the interface but the technology does not add asynchronous response to the client form the application.

Thin clients, here understood as web applications, work in a request – response cycles. A user task trigger that a request is sent to the server. The server processes the request and returns a response back to the client.

2.6 Concurrency and asynchronous architectures

Concurrent programming languages define concurrent actions or processes that may be executed simultaneously. In a concurrent systems the following properties are important assets; processes definition, communication and coordination between the processes, how actions are defined [Mitchell '03]. To achieve concurrency in a system some or all of the following properties has to be available:

- Communication between processes by mechanism as synchronous or asynchronous messaging or shared variables.
- Coordination between processes - may explicitly or implicitly cause one process to wait for another process before continuing.
- Atomicity – basic actions which guaranties atomicity in the interaction.

In addition to be synchronous or asynchronous, messaging can be buffered or unbuffered, and preserve or not preserve the transmission orders of the messages. When synchronous messaging is used the sender waits until the response arrives. In an asynchronous model the sender continuous the processing without interruption.

2.6.1 Java concurrency

Concurrency in Java is achieved by creating a Thread object, by extending the Thread object or implement the Runnable interface, and invoke the run() method. This causes the object to run in a separate thread in the Java Virtual Machine (JVM). Threads in Java communicate by calling methods of shared objects. This communication does not synchronise the threads involved. The Java concurrency model is achieved by three basic mechanisms:

- Lock – a lock for mutual exclusion.
- Wait sets – a thread can suspend itself until another thread awakens it.
- Thread termination – a process can pause until another thread terminates.

Another form of concurrency in Java is when a program running in one JVM is communicating with a program in another JVM.

2.6.2 Actor model

The actor model is a communication form described by Carl Hewitt [Hewitt '76] where objects react in response to messages. Hewitt describes an actor as a “local model of computation” which performs one or more of the following basic actions:

- Send a finite number of messages to Actors.
- Create a finite number of actors
- Designate the behaviour to be used for the next message received.

Actors have no shared objects but uses messaging. Communications with other Actors occur asynchronously. It is a model characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, and interaction only through direct asynchronous message passing with no restriction on message arrival order. The basic idea of Actor languages is that objects have their own threads of control and asynchronous messages sent to them are queued and processed by the object's own thread.

Implementations of the Actor model

The cosmic cube [Seitz '85] is a hardware implementation of a message passing architecture. The parallel processor implementation uses message passing instead of shared values between processes. The architecture are scalable both locally and none locally.

Another implementation that provided architectural support for of the Actor model is the J Machine [Noakes et al. '93] that was developed at MIT. This included the following:

- Asynchronous messaging
- A uniform space of Actor addresses to which messages could be sent concurrently regardless of whether the recipient Actor was local or none local

Concurrent Smalltalk (which can be modelled using Actors) was developed to program the J Machine.

2.6.3 Distributed computing

Distributed computing is decentralized and the processes execute in parallel. The processes are executed by two or more computers and communicated over a network to fulfil a common task. The goal with distributed computing are to create systems that are open, scalable, transparent (the distributed nature is hidden form a user and developers point of view) and fault tolerant.

When a problem occurs the troubleshooting and analysing can become difficult. It might be necessary to inspect and validate the communication and the data on remote computers.

3. Analysis of a Case Management application

Here is an analysis of a genuine full scale Case Management application which is made anonymous. The analyse focus on the user interaction in the application. The goal of this analysis is to get a foundation for a comparison of this reference application with the architecture described in this thesis.

The analysis contains an introduction to the overall architecture of the reference application. It also includes some details in the presentation part of the architecture. The more detailed description indicates some of the problems often found in a typical application with user interaction.

Some of the problematic parts of the analysed reference application are re implemented using the architecture described in the thesis (5 - Re-implementation of the analysed application). The comparison between the two solutions is located in 5.5 - Comparison of the approaches.

Definitions

Here are definitions of some of the concepts used to describe the architecture of the middleware.

- Module - in the reference application, is used to denote a functional unit with a reason to exist by its own. A module the totality of presentation, business logic and interaction with backend systems.
- View - is the presentation part of a module that he user sees and interacts with. (In a user perspective module and view might be grasped as the same)

3.1 Background

A Case Management application handles all aspects connected to the management of a case throughout the case's lifespan. A case is defined as all of the processes, resources, business rules, tasks, and analysis, as well as the details and evidence that need to be captured and managed during the case management process.

The analysed system is a Case Management application for an organization in the public sector. (Hereafter called - CM application) It is used by more than 1000 users and has more than 50 different views. In addition to basic case management it contains task managements for the organization, inquiry resolution, reports, letters, electronic documents etc. The application enforces rules set by the government to pay out and collect financial support.

The CM application is used by the case workers throughout the whole lifecycle from the approval to the continuing pay out of the financial support. The complete lifecycle for a case can last from a year to more than 10 years.

In the usual flow from an application for financial support to approval, the application is scanned and registered in the electronic document part of the application. The registered application generates a task that a case worker set about to deal with. The case worker checks the information and obtains more information if necessary. If all the necessary information is available the case worker continues the process of calculating the financial support. This process has from a couple to nearly 10 steps that the case worker has to go through dependent on the complexity of the case.

The CM application is a complex web based front-end application that uses more than 15 different backend systems and databases with a variety of technologies from IBM mainframes and IBM MQ series [MQ '07] to J2EE and .NET web services. The application was developed with more than 20 developers.

3.2 Architecture

The application uses a common architecture and interaction model frequently used in the industry. It is built using common patterns for Java Enterprise applications (see 2.3.6 - J2EE core patterns) using a web front end. The web front end is using a backend system which is integrated with many other systems in the organisation. The architecture uses many lightweight frameworks to ease the development of the front-end and the integration with backend systems and databases.

3.2.1 Layers in the architecture

A layer is the architectural parts of an application that share the same functionality. The parts in a layer have the same responsibility in an architectural point of view. A layer usually hides its lower layers from the layers above. In contrast to tiers which have a physical separation of parts in the application architecture where different tiers exist on physically separated computers [Fowler '03].

Figure 3-1 below shows an overview of the architecture layers of the CM application.

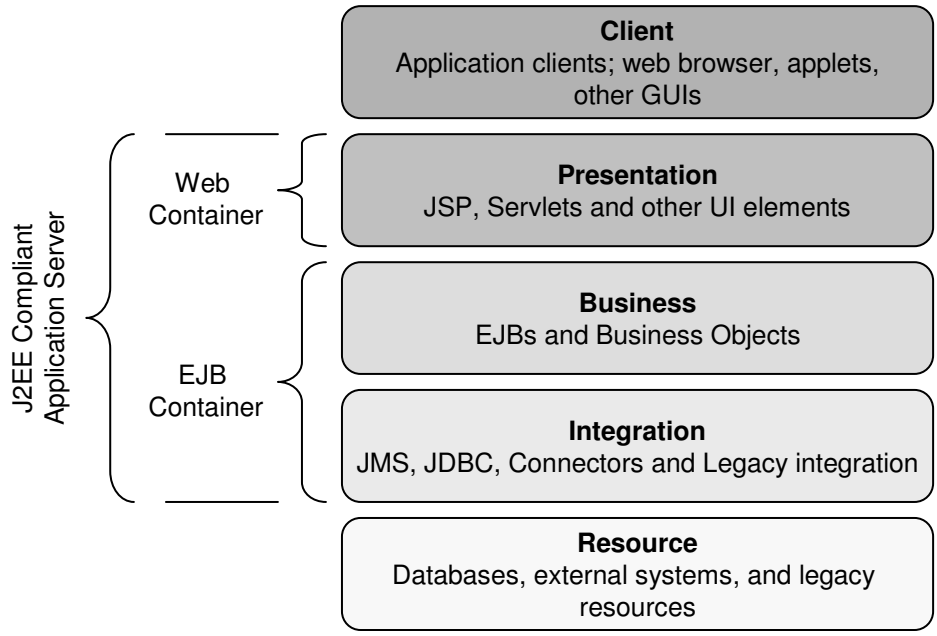

Figure 3-1: Layers in the application

Client

The client is a standard web browser. In addition to HTML Java applets run to extend the functionality of the browser and integrate with the operating system and other applications. There is integration with Microsoft Word for editing and storing letters generated by the application.

Presentation

The presentation layer holds the parts for single sign-on, session and state management, user input validation, navigation, content creation and rendering, format and delivery etc. The layer is executed in a J2EE web container that runs on a J2EE Application Server.

The presentation layer will be described more in detail later.

Business

The business layer has services for mainly business logic but is also linking integration services together. This layer is also executed in a J2EE Application Server.

Integration

The integration layer has resource adapters for connection to different legacy and backend systems with a variety of technologies and protocols. There are integrations from CICS (Customer Information Control System – a transaction server run on IBM mainframe) in a copybook format to Web Services using XML and database integration.

Resource

The resource layer is the collection of all the external system used by the application. The resources offer functionality reused by the organization and handles common tasks. There are also external applications purchased to include in this application to handle functionality that already existed in a standard application. The resources run on IBM Mainframe [Mainframe '07] with databases, IBM MQ [MQ '07] and J2EE Application Servers, and Windows servers with J2EE and .NET Application Servers.

3.2.2 Frameworks used in the architecture

There are some open source frameworks used in the application to ease the development and to "glue" different parts together. Other libraries are also used but these are not important to the analysis.

Struts

The GUI part of the application was built with use of Struts. The Struts framework is one of the most used web application framework for Java [McClanahan '07] [Cavaness '02] [Carnell and Harrop '04] and has been one of the major Java web frameworks since 2001 when it first were developed. It is an implementation of the Model-View-Controller model 2 architecture described in Figure 2-3: Model 1 and Model 2 of web applications. The framework is designed to help developers create web applications that utilize a MVC architecture with a centralized controller. The framework provides a controller component for request handling and integration with other technologies that handle the model and the view part of the application. The controller is the bridge between the application's model and the view. The controller also handles the navigation between views.

In the development of a module in an application the following parts has to be completed:

- Action – An action is a Java class that process the request for a single call by the user.
- Form – A form is simply a Java bean class holding data used by the Action. The Form is populated by the Struts framework and used by the Action and the Java Server Page.
- Java Server Page (JSP) is the rendering technology used to generate the HTML code that is returned to the client. JSP is a template based language for rendering HTML code. The JSP uses data from the Form altered by the Action.
- Struts configuration – A module has to be registered in the configuration with the class definitions (mapping) and the possible navigation form that Action (forward).

A short description of the request cycles in the interaction by a user.

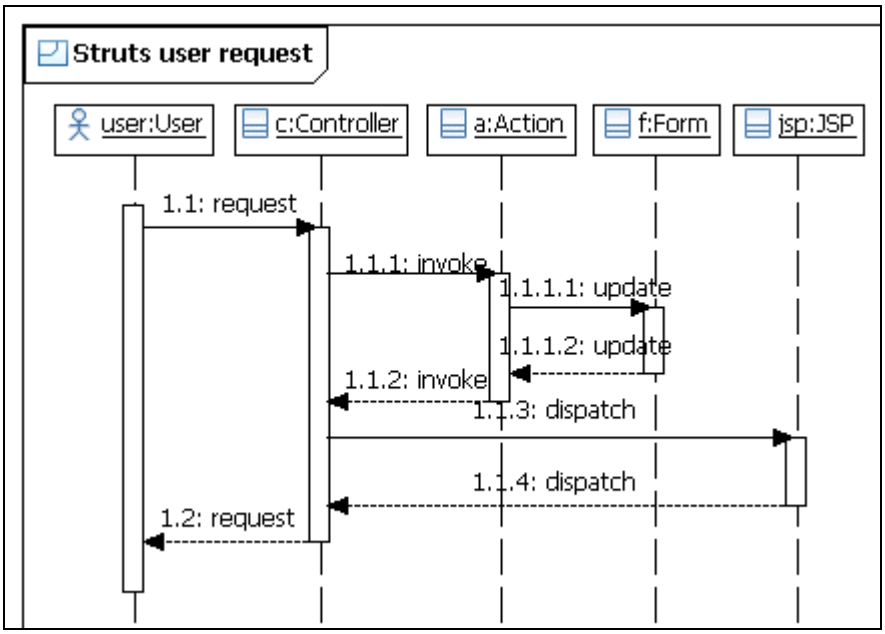

Figure 3-2: Struts handling of a user request

As shown in Figure 3-2 when a request is received, the Struts controller invokes an appropriate Action class located by the mapping described in the configuration file. The action performs the presentation logic that might involve calling the business layer. The action updates a Form with the new information and returns the chosen forward to the controller. The controller forwards to a view, usually a JavaServer Page (JSP), that render the result as a HTML page that is returned to the client web browser.

The navigation between views is handled by the action and the controller. The controller gives the action the information about possible navigation paths for the action. The action decides which path to use based on the result on the computation.

Struts set some guidance of how to develop modules using the framework. The guidance of how to separate concerns between the framework parts as the action, form and JSP. But the guidance is not that firm and it is easy to develop in a way that is not desirable, there are no constraints that forces the developer to follow the guidance. It is up to the developers and the architects to assure that the code base follows a common standard for and are aligned with the principles in the project.

Spring Framework

The Spring Framework [Springframework '07] [Johnson and Hoeller '04] is a Java/J2EE application framework. Spring includes a lightweight container with centralized configuration and wiring of applications using inversion of control [Martin '03] [Fowler '04].It includes framework support for transactions, database connections and object-relational mapping frameworks, aspect-oriented programming [Kiczales '96], web MVC framework.

In the architecture of the CM application Spring is mainly used for wiring the modules and layers together.

Hibernate

Hibernate [Hibernate '07] is an object-relational persistence and query framework including mapping between objects to relational database tables and a query language.

In the application Hibernate is used to map objects to a DB2 database running on an IBM mainframe.

3.2.3 Presentation architecture

The presentation architecture is a J2EE based architecture that uses the previous mentioned frameworks. The Struts framework is the main part of the presentation architecture and handles the request-response cycles. There are implemented extensions to the frameworks to cover the needs of the application.

A request-response cycles

The architecture is a request-response interaction model, using the internet protocol HTTP, where the user submits a request to the server that the application process and returns a response. The interaction model is stateless where no information is, per default, remembered by the server between two user requests.

Description of the request-response cycles in the presentation architecture and use of the architectural components.

1. When a request is received to the Application Server it passes the chain of filters and the context and state information is set. (The context contains user and request information)
2. The input from the user is retrieved and the correct form is populated by the controller. Correct form is figured out with information from the request and Struts configuration files. The controller then validates the form according to the rules set to the specific form. If there are no validation violations the action for this request type is invoked.
3. The action executes the presentation logic for the module that mostly contains one or more call to the underlying application layer. It might also be some presentation actions like adding a new line in a list on the screen or adding fields etc.
4. The Java Server Page renders the html code that is returned to the users web browser based on the information in the form. The JSP contains logic to render different data in different ways. Depending on the complexity of the JSP the rendered pages from one single JSP can have totally different look.

Additional presentation architecture components

There are implemented architecture components for the application to meet the requirements to the application. The ones having interest for the thesis are described here.

Transaction Filter

This is initializing an object representing the context for the request. A context is only valid for the single request. Every request to the server has its own context. The filter sets some information about the user, the module in use etc. as retrieved from parameters from the HTTP request. The state for the next module is fetched from the HTTP request as well. The state information is stored in a regular HTTP parameter. If there is no state information in the request the sate defaults to “normal”.

Controller

There is added some functionality to the default Struts controller. The validation logic has an exception to not execute the validation when a view is loaded for the first time after another view. I.e. the validation runs when an event is submitted. Also parts of the state management which is described below.

Management of session and state

The architecture has added some logic for manage states of the modules in the application.

The struts framework uses the web container built in in-memory session data storage that could be used to preserve state across requests, but it is not used, due to the following reason:

- All data that exists in a view can be stored until the next time the user enters the view again. It is mostly an undesirable solution since the data might not be of interest for the user anymore.
- The data might not be in sync with the persistent storage and thus give a false representation of the information.
- There might also be a heavy cost for the server since all information are stored in the server memory and all users have their own copy until they log off.

Therefore, since the in-memory session storage is inadequate, a component partly overriding the Struts controller is added to do handle the state management. The component set the state in the context (a memory location only reached by the thread handling one user request) before forwarding to the action for the request. This lets the action now in control to choose the state of the following action that is going to be activated. An example, as shown in Figure 3-3 below, a view can decide based on the user event, to navigate to a view in detailed or simple state. The second view decides which state that is going to be based on the information the first view submits.

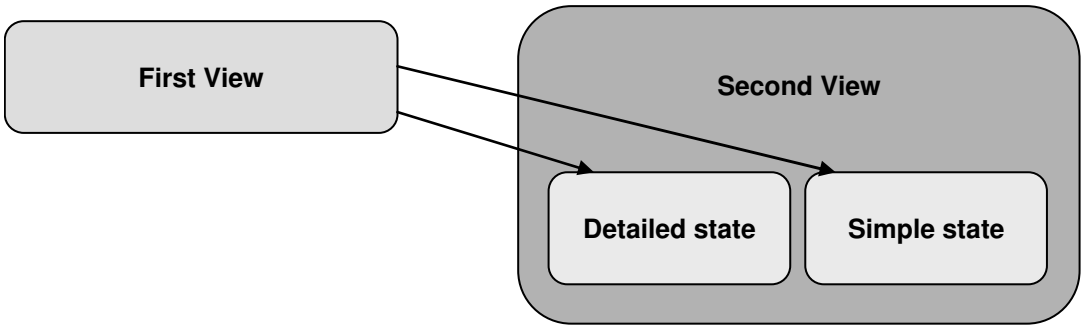

Figure 3-3: View State Management

This is state management is not to be mistaken for UML states as in the UML state machine [Booch et al. '05]. There is no functionality for changing state when executing request in the same view. I.e. a view cannot change state based on an event the user does, without moving to another view. The state does not have different transitions (or possible actions) dependent on which state is active.

3.3 Problems with the application

Here we will lock at some of the problems encountered during the implementation and the maintenance of the application due to the architectural choices.

Many of the problems arise from the request-response interaction model that is in use (described in 3.2.3 Presentation architecture). The request-response does not keep any information between calls from the user. The information has to be stored in the session context of the web container. The Struts framework (3.2.2 Frameworks used in the architecture) implements the request-response interaction model and some additional architecture elements were added to cover the functionality needed for session management in the application (described in 3.2.3 Presentation architecture).

The problems have a short description and analyse.

3.3.1 State management

The management of the different states a module can have in the application. For example if a module is in a detailed or general state the number of elements shown could be different.

The problem in the application was that a module could not have internal states. The need for state as described above with the detailed and general module had to be implemented using custom code in each module. Conditional logic had to be used, in the JSP part of the module, to decide if an element should be shown or not. Variables had to be added and maintained in the logic, data carrier and the presentation part of the module (Action, Form and the JSP). If a module had requirement for a couple of “states” the conditional logic become complex and difficult to maintain.

Management of the state in the modules of the application were quite coarse-grained and not flexible in use. The functionality was far from the UML state machine definition. The state of a module could just be chosen when entering a module and could not be changed when using the module as illustrated in Figure 3-3.

The modules are per default stateless. When a user event triggers an action the action starts with no information of previous state. And when the action has finished no state information is remembered. If the module needs to remember a state the developer has to take care of that. The state has to be saved in the web containers session or persisted in a database. Saving state in a web container is a cause to problems because other modules could use or change the same information resulting in changed or lost state information.

There are no methods for informing other modules of a state change. If a state change in one module should influence some other modules this has to be implemented in the affected modules separately.

3.3.2 Navigation

Navigation is moving between different views in an application using the paths which is defined for connection between the views. The navigation can be in a high level, between different functional areas, or low level, in a search-result loop.

The navigation in an application using the Struts framework is split in a centralized configuration and a local module decision.

If the navigation has to change the changes is performed both in the action and in the central configuration.

One configuration file knows all the possible navigation. The configuration has information about the possible navigation every action can take, which view is the next etc. This is like one big state machine controlling the navigation where the action is responsible for choosing the next navigation path that has to be defined in the configuration.

This is a coarse grained approach where the responsibility is split between the framework configuration and the action. There is no functionality for splitting the navigation in smaller peaces for parts of the application.

In the CM application there are a number of places where the user can choose the navigation path. Different paths depend on the information that is available for the case worker. One path might be an information search and gathering that returns to the same point in the overall navigation.

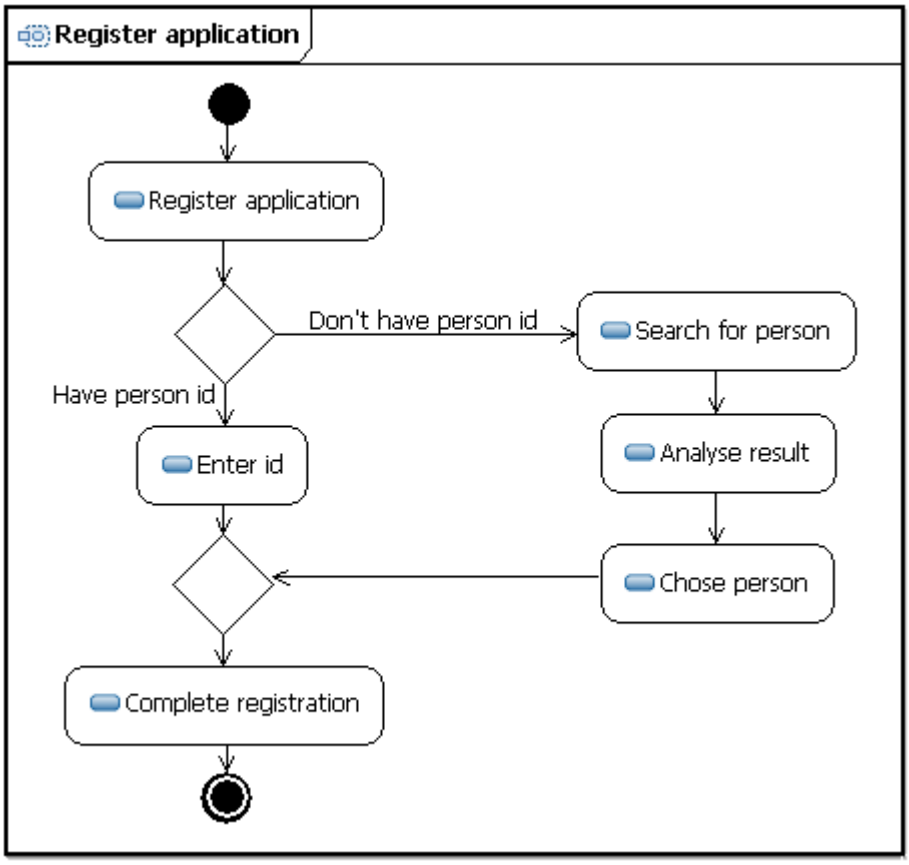

Figure 3-4: Search during application registration

The search path takes control of the navigation during the search and the user can navigate back and forth between the search view, the result view and the detailed view before navigating back to the main path as seen in Figure 3-4. A separate control and state management for the search would have been desirable but the Struts framework does not have this support.

A fine grained management for navigation in sub paths where the state of the main path is remembered would have been ideal. If the sub path had to change the main path should not be needed to be changed.

3.3.3 Long running transactions

Some steps in the process of the application have long response times due to usage of many backend systems. The problem is that the current step in the process has to be finished before the user can continue with the next step. This step halts the user's interaction with the application.

The parts of the CM application that is using external backend systems have longer response time than the parts using the database owned by application. There is a considerable latency that is easy to notice for the user. To rescue the latency caching has been added to the application in a number of places where a backend system is used. This helps the respond

time and makes the application more respondent. But in some situations caching can not be used since the actual result of the response is needed to continue the navigation flow. In these situations the user has to wait from a couple of seconds up to a minute in extreme situation. All the communication with the backend systems in the application is synchronous.

3.3.4 Notification

The application has notification of information that is of interest for all users of the application. The problem is that there is no possibility for submitting the notification to all users at theirs position in the application (the users current active module).

The CM application has the ability to show notifications of administrative information in a part of the main view where the case worker has the task listed. The notification is typically about maintenance regarding the system or downtime information. The notification is only shown in the main view so when a case worker is in a process working with an application, which might last for an hour, the worker does not see the notification that even might affect the ability to fulfil the ongoing task. This might be frustrating for the worker and time is wasted.

3.3.5 Misuse of the architectural instructions and processes

The architectures and the Struts framework have guidelines of where to put business and presentation logic for an optimal organisation (described in 3.2.2 Frameworks used in the architecture). The organisation is useful both during the implementation and the maintenance of the application to help and guide the developers.

The guidance has no constraint that has to be followed; the runtime does not force the guidance. The implementation does not follow the guidance. This has caused a fragmented code-base that is difficult to read and maintain.

When many developers work in the same project and in a short time frame the implementation of one module are often marked by the developer fulfilling the task. The modularization is not always optimal and does not always follow common principles and standards.

Another problem is that the guidance not always fully covered. There are always areas not thought of while describing the guidelines and that might create confusion and result in diverse solutions.

Examples of misuses of the guidance and principles:

- Presentation logic is placed in the Form or JSP instead of in the Action. Some of the presentation logic has to go into the JSP but just the part that has to do with the rendering of the response.
- No strict rules for placing of validation logic. Validation of user input is be performed both in the action, in the form and by using configuration files.

- Business logic is performed in the Action that should have been moved further down in the architecture layer.
- Modularization that were in an inappropriate level; complex inheritance hierarchy, too large modules etc.

3.3.6 GUI change

The views in the application could be changed due to actions taken by the user. Fields, buttons or other widgets can be added or removed. In the reference application this change has to be handled by the code in at least three parts. This adds complexity in a maintaining perspective when a single responsibility is implemented in different places.

The GUI widgets in the reference application are rendered by the JSP. The JSP is using conditional logic base on information in the data holder (form) to validate if a widget should be present or not. The logic for the decision on if the widget should be present is located in the action that is responsible for the user action. This gives that three parts has to cooperate to handle the dynamic GUI change of a view. And the fact that the third part, the JSP, uses procedural logic to decide if the widgets should be present or not makes the JSP complex if more than a low number of widgets has this dynamic presence. If the number of widgets and states that the JSP has to handle rises the JSP is much more exposed to errors.

4. Presentation Middleware

In the following chapter we will look into the middleware solution for user interaction that is proposed by this thesis. It is a solution that prevents many of the previous mentioned problems with state management and asynchronous processes in GUI applications and has some additional interesting features.

4.1 Introduction

We will start by looking at the overall architecture and continue with examples of implementation before examining the principles behind the solution.

Definitions

Here are definitions of some of the concepts used to describe the architecture of the middleware.

- Part - is used to denote the smallest logical concept in the middleware architecture (Presenter – Controller – View as described later in this chapter). It can contain the logic for a number of GUI widget or the logic for a single button. A part has connections to the parent and the children, if any. The part may have a user interface (view) but it is not required.
- Module - is used to denote a more coarse grained concept then a part. A module is a functional unit with a reason to exist by its own. A module could be a whole view that is used to interact with the application or a collection of smaller parts. (See the module definition for the reference application in 3 - Analysis of a Case Management application)
- Events - are user interaction events triggered by the GUI. A user event is captured by a “listener”. A listener is a piece of code that is attached to a GUI element using the observer pattern [Gamma '94]. The essence of the pattern is that one or more objects can register to observe an event that might be raised by the observed object (Here: user events triggers notification of the observer).
- Signal - is a container for messages. The signals are arranged in a generalization tree. Information to the parts in the architecture is transmitted as signals.
- Java Virtual Machine - JVM - is used in the description and refers to the runtime platform used. The JVM is used to describe the runtime in the illustrations. An application implemented by using this architecture can be running in more than one JVM.

4.1.1 Localization of the Middleware

The middleware is a layer between the actual user interface and the services of the application. The middleware is a messaging layer that transports signals between different parts of the application. The layer also keeps track of the states of the parts and controls the state of the actual user interface.

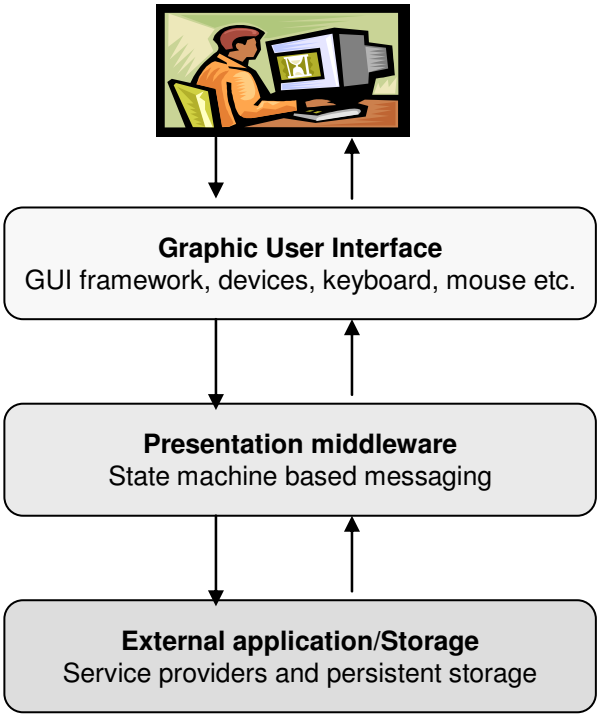

Figure 4-1: Localization of the Middleware

As shown in Figure 4-1 the graphical user interface framework is the link between the physical devices and the architecture. User actions are transformed to events that the application uses. The middleware routes signals through the part hierarchy and the receiving parts react to the signal. A signal could be submitted to a service or storage layer which stores the information carried by the signal.

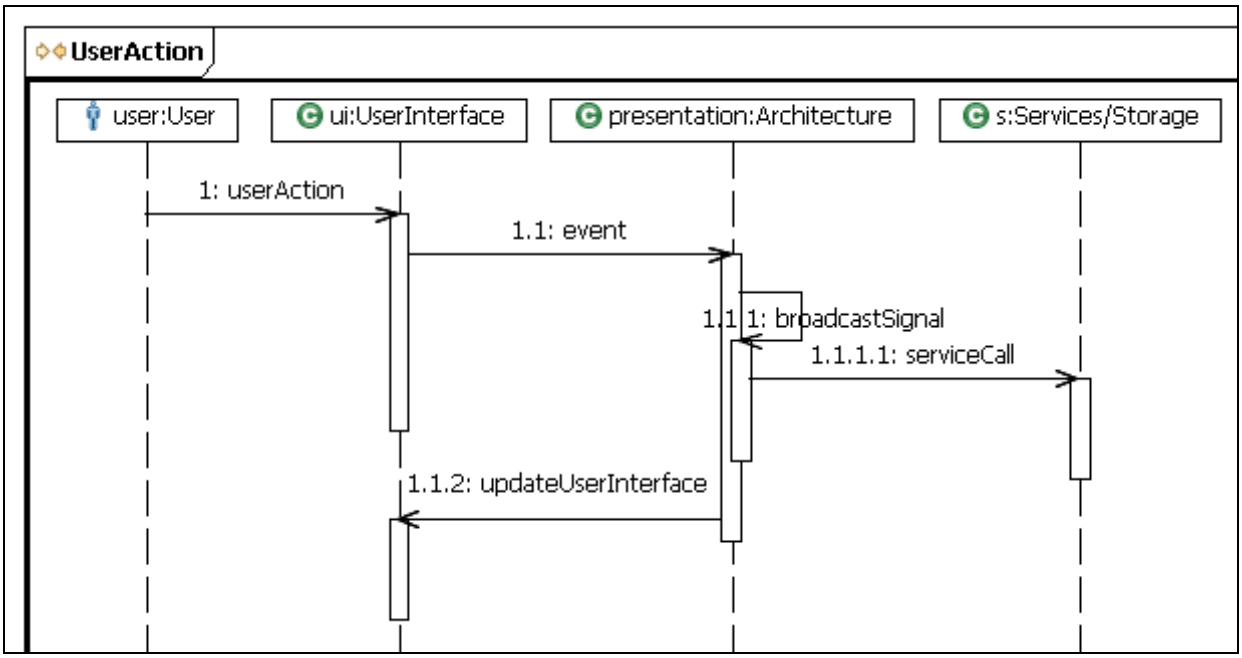

Figure 4-2: A user action in the presentation architecture

Figure 4-2 above, illustrates a user interaction with the user interface and the presentation middleware. The user interface can be anything which can interact with a computer or the network. Usually in applications the user interface is a graphical interface managed by the libraries which control the devices and screen rendering like Java's Swing or SWT. The user interaction triggers events in the presentation architecture layer which in turn submits signals. A controller handles the signal and broadcasts the signal. A service receives the signal and stores or process the information.

4.1.2 Features of the Middleware

The most central features of the presentation middleware architecture are:

Asynchronous messaging

A central feature of the middleware is the asynchronous messaging of signals between the different parts in the application. Signals are forwarded to the appropriate parts of the application and the execution of the signals is conducted in parallel.

Modular

The parts of the architecture in the middleware is built and assembled following a strict structure. The parts consist of one presenter, a controller and a view. The presenter-controller-view (PCV) model separates the responsibilities of communication, state management and user interaction to the components in the model.

Hierarchical structure

The parts of an application built with this middleware are connected to each other in a hierarchical manner. One part of the application can be connected to one parent part and many children parts. User and system events are broadcast asynchronously throughout the hierarchy to every part of the application.

State machine based user interaction

Another important feature is the controlling of the interface that is performed by state machines. Every part in the application has a controller. The controller is a specialized state machine for use in this middleware architecture. The state machines are executed by the runtime of the middleware.

Broadcast bouncing

One central characteristics of the architecture is the broadcast bouncing of signals. The signals are distributed throughout the hierarchy, first upwards to the top and back again to the end nodes so every part has the opportunity to react to the signals.

Model Driven Architecture approach

A system developed using the middleware is by advantage modelled using UML and then transformed to a running implementation. This is a Model Driven Architecture (MDA) approach [Frankel '03] where the model of the system is independent of the runtime platform.

4.1.3 The middleware and Model-View-Controller

The middleware implements the MVC architectural pattern, 2.3.2 - Model-View-Controller. As described, MVC can be implemented in different ways with a variation in the responsibility of the different parts.

The view in the middleware has some similarities to the MVC view described by Reenskaug. The view could be one of many components displayed on the screen. But it is different in accordance with the connection to the model. The view in the middleware is fully controlled by the state machine based controller in the presenter-controller-view triplet. The controller updates the view and responds to events triggered in the view component. This makes the view in the middleware simpler than the implementation described in Sun's Blueprint (Figure 2-1).

The controller is not controlling the whole application as usual in the Model 2 implementation of the MVC in web architectures (see Figure 2-3). It is a UML state machine which controls and keeps track of the state of one view only. The controller in the middleware is executed in separation form other controllers in one application, the connection between controllers is accomplished by asynchronous messaging.

The model in the middleware is represented by the information passed by signals to the controller. The signals reflect different views on the domain model, as stored in a backend system, and not the total model of the application. Querying of the model is accomplished by submitting signals in the middleware which is responded by another part in the hierarchy. A change in the model can be notified by a submission of a signal to all parts in the middleware.

The hierarchical structure of the middleware is close to the conceptual architecture of the PAC model, described in 2.3.3 - MVC variations. The middleware have a strict implementation of the controller and the data propagation mechanism which the PAC model lacks.

4.2 Architecture overview

The presentation middleware is based on two important parts from UML [Rumbaugh et al. '04] [omg.org/uml '07]: composite structures and state machines. These are central to the architecture and are used for communication and state management. The different parts of the application are organized in a hierarchical structure and the communication between the different parts is asynchronous. Changes of the state in one part of the architecture might cause a broadcast of a signal throughout the architecture.

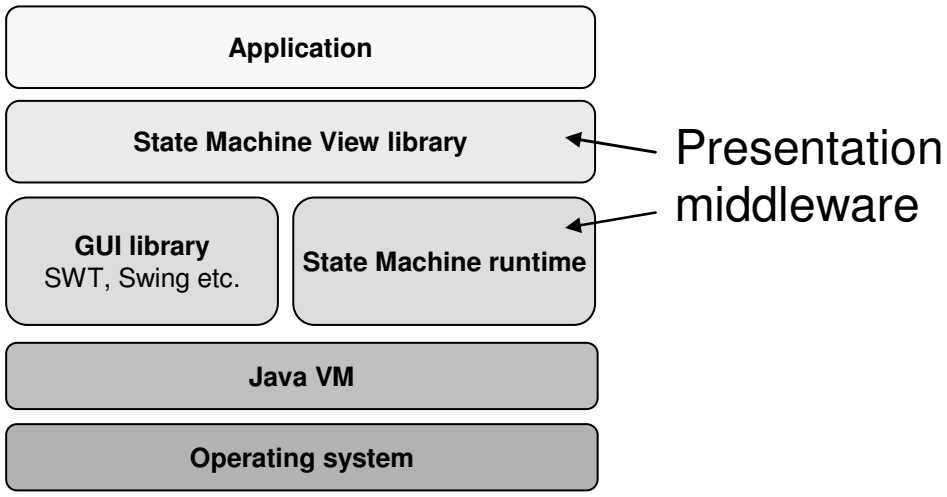

Figure 4-3: Layers in the architecture

The architecture consists of a runtime engine (State machine engine in Figure 4-3) which executes the active components in the architecture (the controllers). There is a library for building view components and adding the communication and interaction with the GUI environment. Figure 4-3 shows the layers for the architecture implemented on a Java platform with the Java Virtual Machine (JVM) as base runtime.

The architecture is presented with examples from a stopwatch application implemented using the presentation architecture.

The architecture and the examples are implemented in Java but the architecture is independent of the runtime environment and programming language. There is also a discussion around the implementation later. UML is used to describe the architecture.

4.2.1 Structure of the architecture

In Table 4-1 there is an overview of the important parts of the architecture and a short description before they are described more in details.

Table 4-1: Structure elements of the architecture

Architectural element	Description
Presenter	The presenter is a specialized UML composite structure that consists of a controller and possibly a number of children. The children are also presenters. The parent-child structure is a hierarchical structure.
Controller	The controller is a UML state machine. The controller handles the state changes and reacts to signals.
View	The view implements the interaction interface. The view contains the elements that are displayed on the screen. It could be a simple button or a complex composite component with many widgets.
Mediator	The mediator is a UML port that passes signals to another mediator (or some other receiver like a controller) connected to it. The basic mediator is connected to one other mediator. The mediator might have a filter attached that instructs it to only pass signals based on the rules in the filter. There are some specialized mediators, for example a mediator for broadcasting a signal to many other mediators.
Signal	A signal is an object which contains the message to the receiver. A signal has no behaviour on its own it just contains information. Signals are forwarded through mediators.

A system is created as a UML model which is later transformed into an actual implementation using the architectural framework. The controller is a State machine and the mediator a Port as stated above. The presenter is a class with a composite structure diagram describing the connections. The view could be modelled as an interface or a class. All elements are used in the modelling of the application.

4.2.2 Details of the structure

Figure 4-4 shows the relationship between the structural parts of what makes up the presentation architecture.

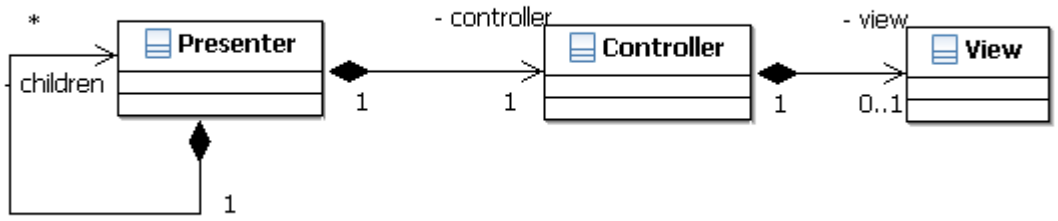

Figure 4-4: Presenter - Controller - View relation

The presenter, the view and the controller work in triplets. Together they act as one part in the architecture. This is the basic building block which is used to build an application. A presenter and a controller can work without a view if the part does not have any interaction except for the hierarchy connections.

Presenter

The main part in the architecture is the presenter. The presenter concept is used as a container for a controller and the children of the presenter. The internal communication between the presenter, the controller and the children is accomplished by connections of mediators.

Controller

The controller owned by the presenter is essentially a UML state machine. The controller is in charge of the behaviour of the presenter and the view the controller might have. If the presenter has any children the connection with them is through the controller by connections from ports the controller has. The actions the state machine executes, in the state transitions etc, can result in submitting a signal through one of the ports the controller has. The controller has one port directed to the owning presenter and one port towards the children. The controller owns the view and handles events triggered in the view.

View

The view represents the user interface part that interacts directly with the GUI libraries. The view consists of GUI elements like text fields, buttons, images etc. The view also defines the interface the controller has access to which includes getting and setting values in the interface and adding event listeners.

The layout of the widgets in the view is not managed by the middleware; it is the responsibility of the view.

Presenter – Controller relationship

The presenter is a container which has a controller for handling the behaviour of the part. The presenter owns the controller and is connected to the controller through UML ports and UML connectors. The signal a presenter receives is forwarded to the controller through the mediator that connects the presenter and the controller.

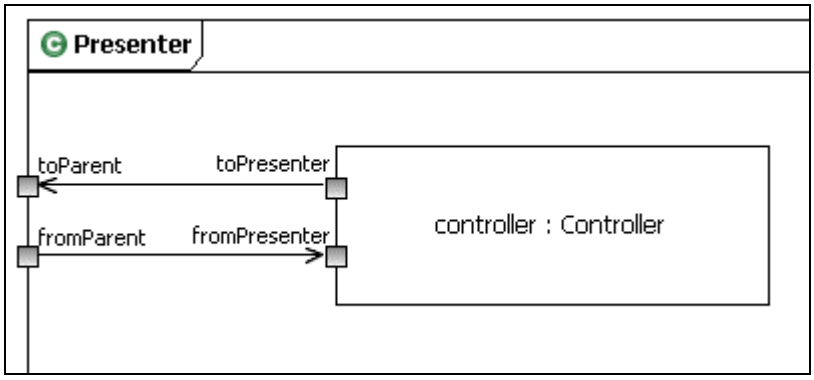

Figure 4-5: The presenter controller composite structure

In Figure 4-5 the presenter and the controller have a two way communication canal where signals can pass in either direction. After the creation of the controller there is no direct invocation from the presenter to the controller. The presenter is not active in the execution after the creation.

Presenter hierarchy

The presenters are organized in a hierarchical structure with a presenter on the top. The connections to the children are established through two mediators; one for receiving signals from the children and one for sending. The controller uses a broadcasting mediator for sending signals to the children.

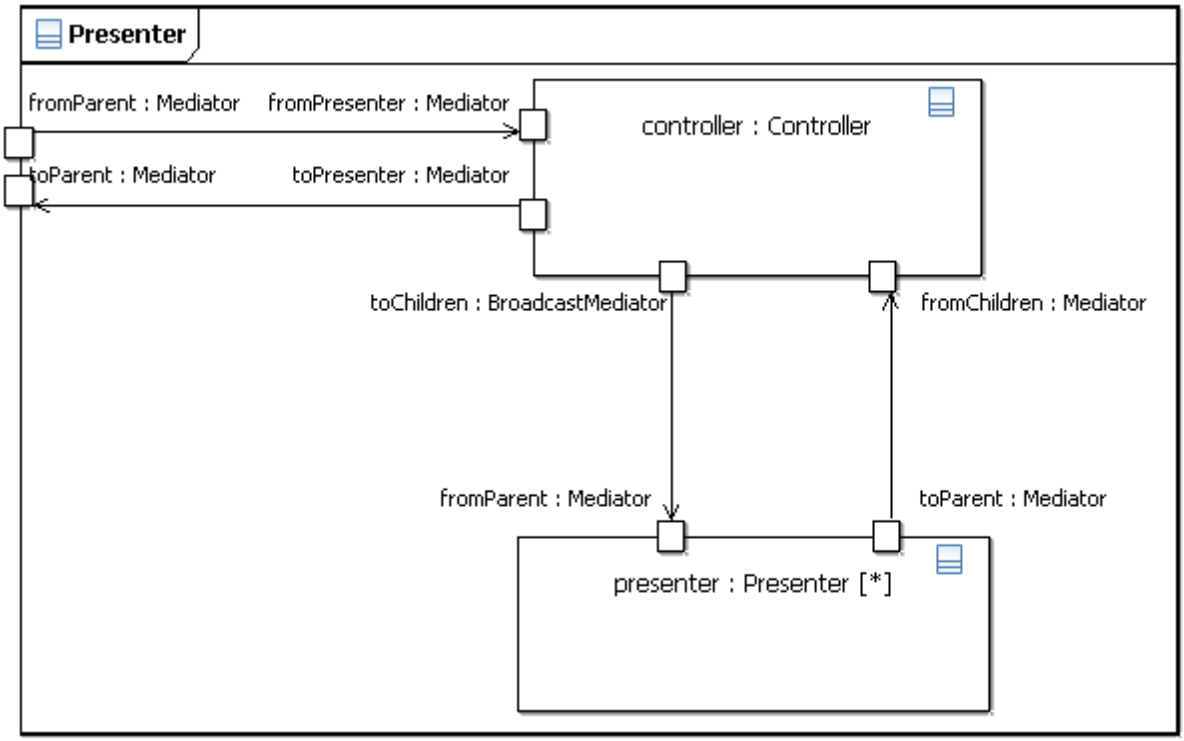

Figure 4-6: The presenter - controller hierarchy

Figure 4-6 shows the presenter children located inside of the presenter parent. It is a recursive structure of presenters where one presenter can have zero or many children and zero or one parent. This structure is like the Composite pattern [Gamma '94] for representing part-whole hierarchies.

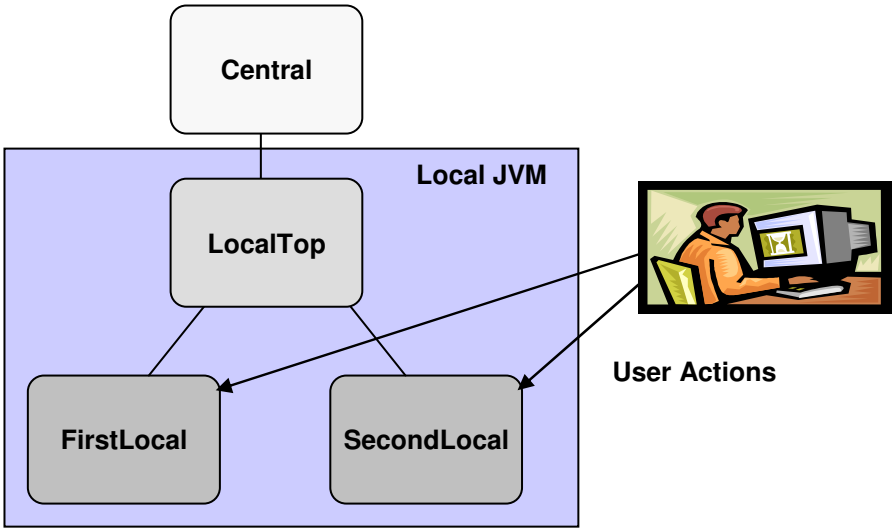

Figure 4-7: Example structure of a hierarchy

Figure 4-7 shows an example hierarchy structure an application can have. Here there is a central presenter as the top of the application which bounces the signals to the local children. The central presenter offers children to connect at runtime and the central presenter can have zero or many children. The application spans over multiple Java Virtual Machines. The central presenter acts as the top presenter for the application and the message passing and

behaviour of the application is not affected by the distribution of the application. The local top presenter owns the GUI in the local JVM and creates the children, first and second, presenters. The first and second presenters add views to the GUI and can be triggered by user actions.

Controller – View relationship

The GUI element, which is visible to the user and has the event handling functionality, is located in the implementation of the view. The view implementation holds the event handlers that the operating system/GUI library uses to inform the application of user actions.

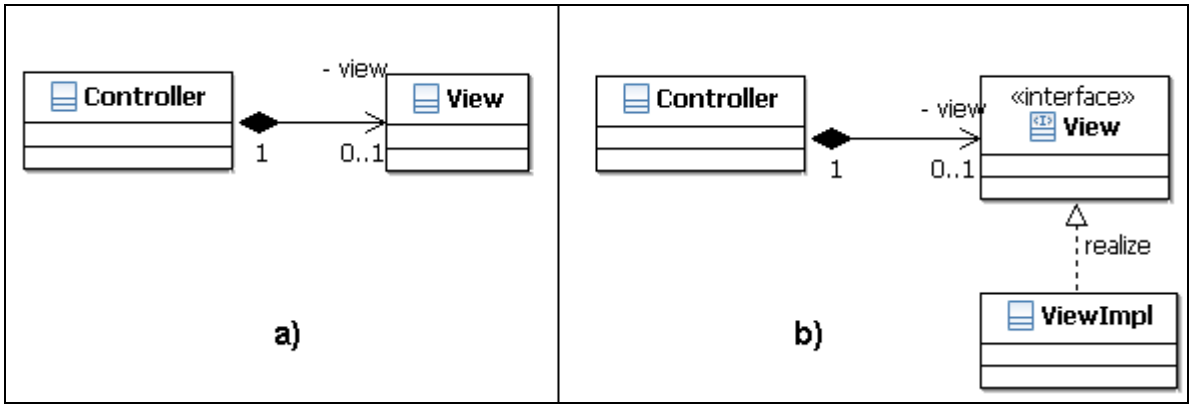

Figure 4-8: Controller - View structure

The first part of Figure 4-8 shows an implementation where the controller holds a reference to the view. The controller is also responsible for creating and handling events that the view receives. The controller accomplishes this by creating event listeners and attaching them into the view. Only the controller has the privilege to change the view, no other part of the application has the ability or knowledge to alter the view.

The view has no dependency to the controller. That makes the view loosely coupled from the controller. This follows the simple Dependency – Inversion principle that states:

”A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
B. Abstractions should not depend on details. Details should depend on abstractions.”
[Martin '03]

Using this separation, with an interface defining the internal protocol between the controller and the view, makes the logic and the protocol easier to test and maintain. This is a driver for putting the logic into the “smart” object, the controller, and have a clean and simple view as the Humble Dialog Box [Feathers '02].

If different appearance of view is required, having the same functionality, the view can implement an interface that the controller holds reference to (second part of Figure 4-8). Then views can have different look but use the same presenter and controller, the same

internal protocol between the view and the controller is used. The presenter and the controller are reused.

Handling events

When an event is triggered by a user, the listener, created by the controller, reacts and submits a signal to the controller to handle the situation.

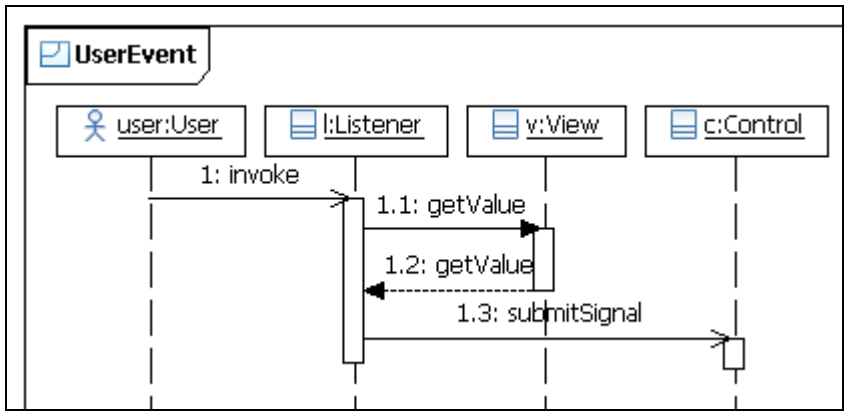

Figure 4-9: User triggers an event

Figure 4-9 illustrates the interaction when a user is pressing a button. The listener, attached to the view is triggered by the GUI framework by standard call-back functionality. The listener collects the information needed from the view and submits a signal to the controller. The listener does not do anything more that creating and forwarding the signal.

Other usages of the abstraction

The interaction does not necessary need to be an ordinary mouse or keyboard interaction performed in a GUI. Interaction with any kind of device can causes the event that leads to a broadcasting of a signal. The approach described here, with a controller – view relationship (see Figure 4-8), is used in the same way but instead of controlling a view a device or a sensor could be the interaction part.

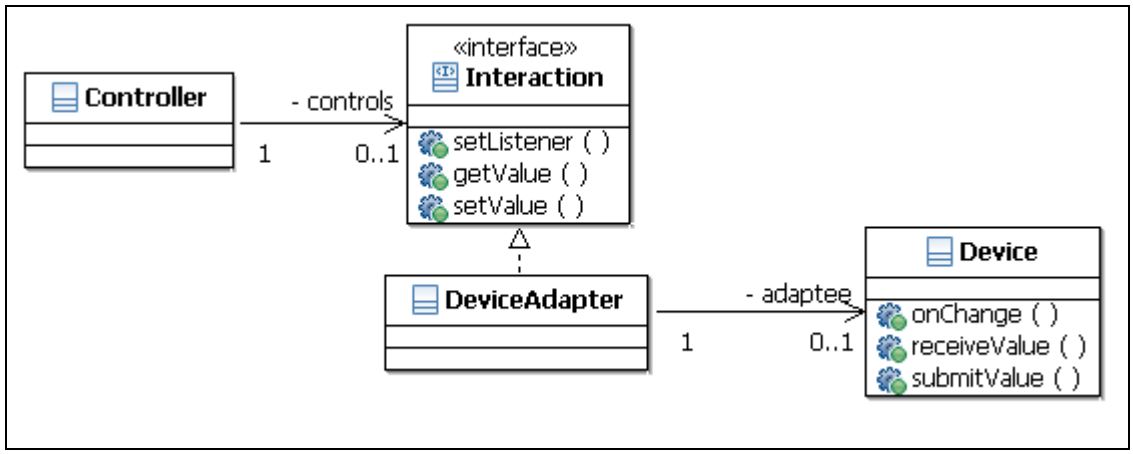

Figure 4-10: Adapter – Controller structure

Figure 4-10 shows the implementation of the interaction adapts to a device and let the controller take care of the state management and communication in the application. The adapter class is specialised to handle the specific device or sensor but the other parts of the application need no customization.

For example if an adapter for a medical monitoring device implements the interaction interface, the controller can broadcast signals about the state of the patient being monitored. The abstraction makes the underlying architecture transparent to what kind of interaction device that is used to listen to events.

Addressable

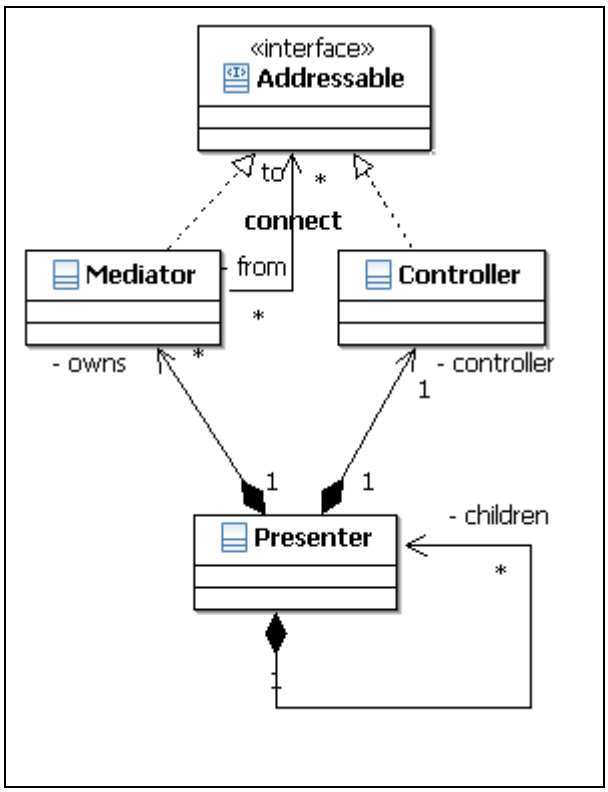

Figure 4-11: The Presenter Controller concept model

As Figure 4-11 shows the mediator can have a reference to any Addressable object: a controller or another mediator, which signals can be passed through. This shows how the mediator is the connection between the presenter and the controller.

4.3 The presenter protocol

Here follows a description of the behaviour of the middleware. The focus will be messaging and control of the interface, but also on dynamic behaviour and extension mechanism.

4.3.1 Broadcast bounce

As mentioned earlier, broadcast bounce is a central concept in the architecture. This is a feature where signals are submitted to the parent of one presenter and broadcasted to every part in the application through the hierarchy. The signal bounces to the top presenter and is forwarded downwards to every presenter node in the hierarchy.

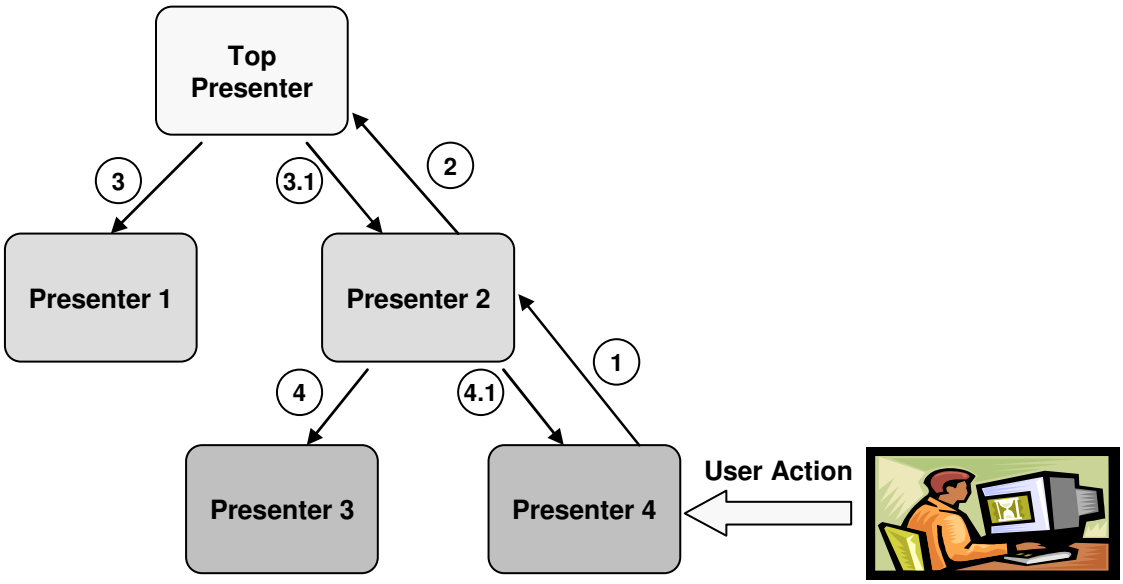

Figure 4-12: Presenter Hierarchy signal bouncing

Figure 4-12 is an example of a signal bouncing when a user triggers an event in presenter 4. The event causes the controller to perform a transition and the action invoked submits the signal to the parent. The signal then passes throughout the hierarchy and finally returns to the presenter submitting the signal. When the signal passes a presenter, the presenter can take action according to the signal or just pass it on.

The middleware implementation executes the signals by the controllers on the way up as well on the return from the top presenter. The controller is not aware of where the signals have arrived from or where they should be forwarded to. The signals are placed in the same input queue independent of the mediator they arrive from. The controller does not have any special functionality for participating in the presenter hierarchy more than having four mediators connected. The actual logic for broadcasting the signals in the hierarchy is implemented as actions in transitions between states in the controller.

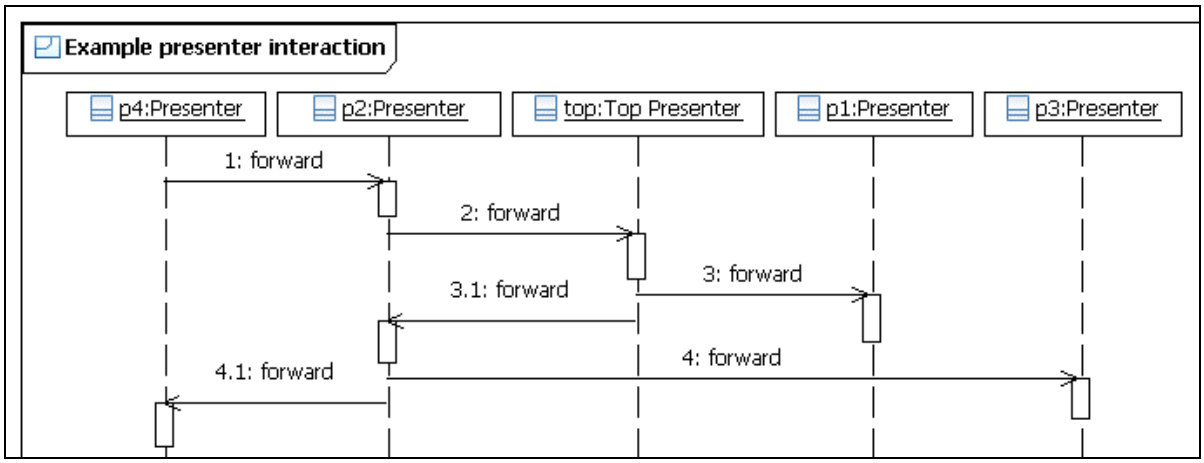

Figure 4-13: Example presenter interaction

Figure 4-13 shows one possible signal flow in the hierarchy. The signals are forwarded to the next presenter level. When the signal reaches the top, it is broadcasted back to the child presenters of the top presenter (3 and 3.1). Presenter 2 has own children presenters and forwards the signals to them.

The sequence of the forwarding of the signals is not guaranteed to be in the sequence shown. Due to the asynchronous nature of the middleware runtime the controllers of the presenters are executed by the engine in any order. For example, the message 3.1, 4 and 4.1 could be completed before message 3. But the signals in a single controller are guaranteed to be executed in the order they arrive.

This behaviour is the foundation for giving applications built by this middleware, a modular design in respect of simplicity and loose coupling to other parts. A part has a coupling to other parts only by the signals which are received. This gives the presenters a single responsibility for their own state and transitions. A presenter is only responsible for their own behaviour and not for the others.

Constrained broadcasting

The bouncing does not have to go to the top in all cases. An application could have an internal top presenter that bounces the signals to the child presenters. The internal top could let some of the signals through to its parent or might collect a number of signals and submit the collection when it reaches a given number.

A presenter could collect information from sensors of some kind and transmit the information when a given number of signals are received or if the values in the signals pass a given boundary.

4.3.2 Asynchronous communication

In the middleware, the asynchronous delivery of messages and parallel execution of the controllers are important features. When an action, either in response to a user action or in a state transition, submits a signal, the delivery of the signal and the processing of the signal

occur asynchronously. The signal delivered might be positioned after a number of other signals which are to be processed by the state machine before the newly delivered signal. The processing of the signal might also be executed by another thread than the thread which delivered the signal. The engine uses a numbers of threads to execute the controllers. The engine consists of a number of workers that owns the threads and executes the controllers.

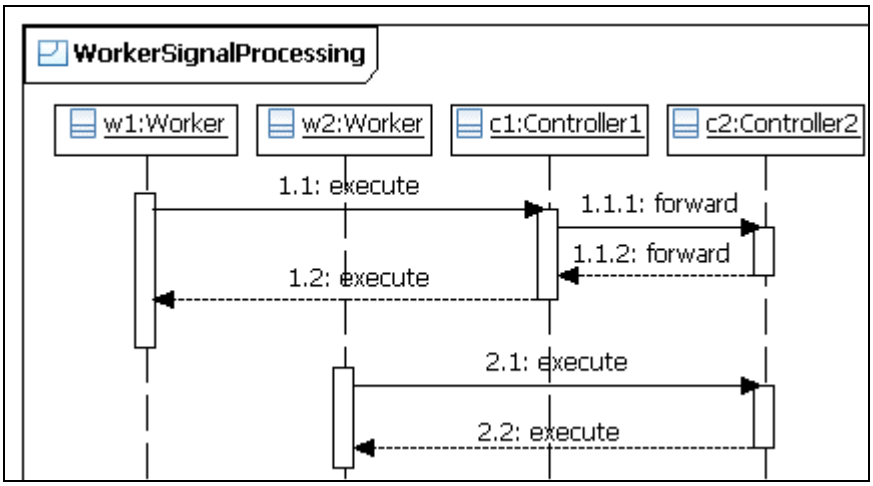

Figure 4-14: Processing of controllers

Figure 4-14 is an example of how the execution of two controllers could be performed by two workers (w1 and w2). The workers run in separate threads by the state machine engine. The first worker executes the first controller which fires a transition. This transition has an action that forwards a signal to the second controller. Finally the second controller is executed by the second worker.

Execution of View Actions

It is not only the threads owned by the state machine engine that is running in an application using this middleware. There is also a thread owning the GUI; the Event Dispatch Thread (see 2.4.2 - Standard Widget Toolkit). Only this tread can access the GUI widgets of a view. The middleware defines a special action, view action, for updating the GUI. The actions created as view actions is executed by the Event Dispatch Thread.

4.3.3 Controlling the interaction interface by state machines

The interface is controlled by state machines. In the middleware a controller is a state machine with some additional features. The controller has incoming and outgoing mediators to the owning presenter and the children of the presenter. A controller usually also owns a view. (This might not be the case for the controllers owned by the central presenters)

To handle user actions the controller injects listeners into the view that submits signals to the controller in case of an event.

Here follows an example of a state machine controlling a view. This is an example from a stopwatch application with common stopwatch functionality as starting, stopping and

clearing the time and also suspending and splitting the time. This example is described in more detail in the section 4.4 - Example of use.

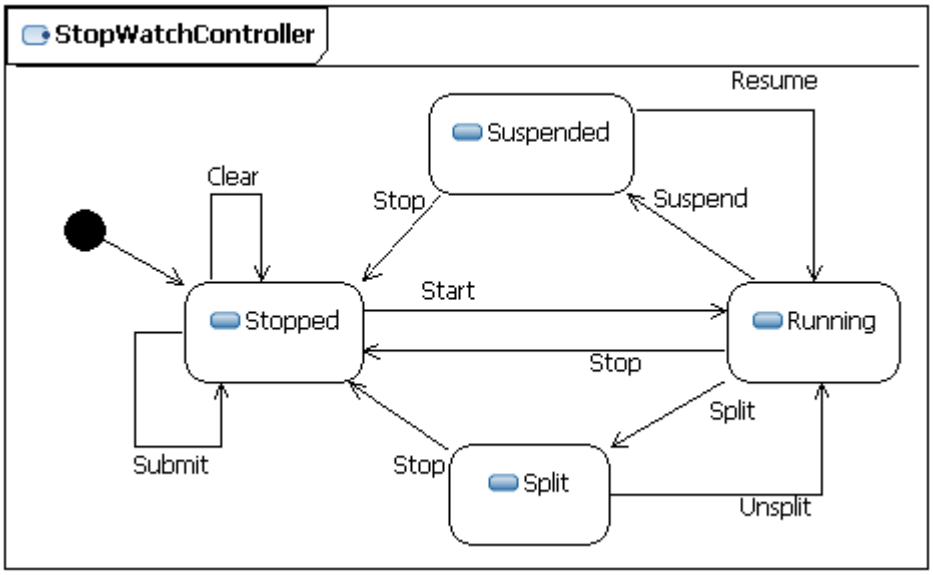

Figure 4-15: The stopwatch controller

Figure 4-15 shows the state machine for the stopwatch user interface. When the stopwatch is running is can be stopped, suspended or split. The submit transition forwards the result in the hierarchy of the application.

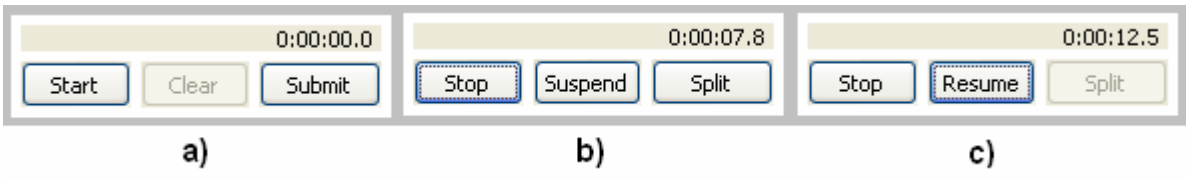

Figure 4-16: Three appearance of the stop Watch user interface

Figure 4-16 shows how the user interface looks for the stop watch in some of the states. When the stop watch is in the Stopped state (a) the Start button is enabled but the Clear button is disabled. When the stop watch is running (b), the Start button has changed name to Stop and the Suspend and the Split buttons are enabled. In the Suspend state (c) the Suspend button is renamed Resume and the Stop button is still enabled.

The changing of the user interface in the stop watch is performed in the transitions between the states in the controller.

4.3.4 Filtering of signals

With broadcast bounce every part of the application receives the signals which are submitted. It might be a lot of signals which are passed through the presenters and they might not be interested in all the signals passing. Therefore there is functionality for filtering signals in the mediator where the signals pass. The filter has rules for accepting or rejecting signals.

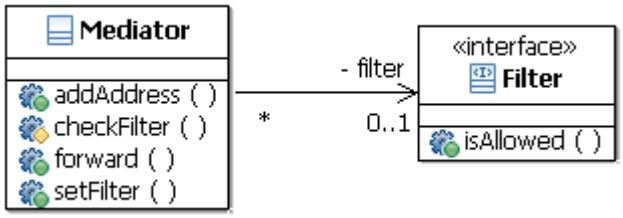

Figure 4-17: Mediator and Filter

Figure 4-17 shows the class diagram for the mediator. If a filter is added to the mediator the filter is checked by the mediator before forwarding to the receiver.

There is an example of filtering signals in the stopwatch application described in section 4.4.5 - Filtering of signals.

4.3.5 View extensions

A view extension is a point of which other parts can use for presenting their view elements. A part in the architecture can offer a view extension that a child presenter can use for publishing its view elements. The extension could be a number of different parts in the GUI which the children can use for adding their behaviour and elements. Examples of view extensions are a tab folder, menu bar, status bar, composite structure (an area in a GUI window) etc. A child presenter can offer its children other view extensions to use.

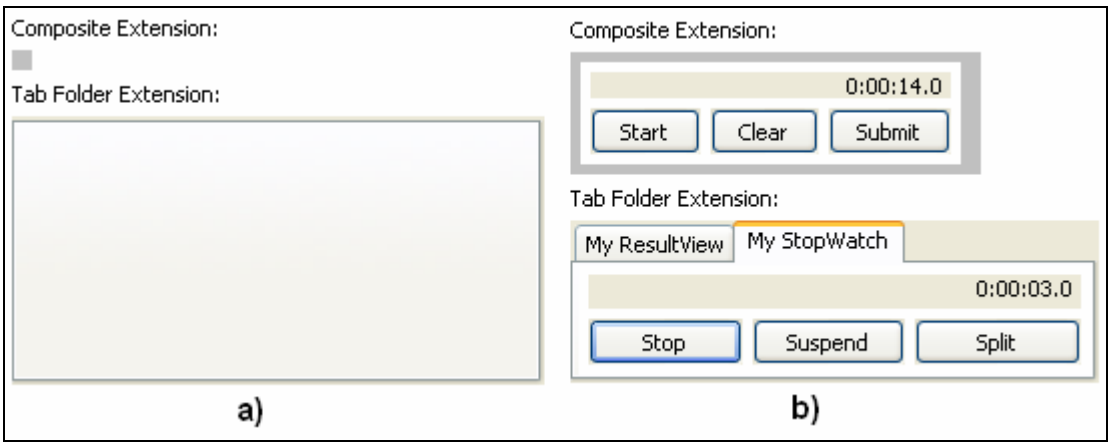

Figure 4-18: The stopwatch extensions

Figure 4-18 shows an example from the stopwatch application, described below in 4.4 - Example of use, where the top presenter offers a composite extension and a tab folder extension. The top presenter has no functionality or view elements added to the extensions, and, as shown in the first view of Figure 4-18, the view is empty without children. The children of the top presenter add the logic and the GUI elements. In the second view of Figure 4-18 three children has added their views; two stopwatches and one result view.

The view extension mechanism gives the middleware a dynamic approach to adding GUI and logic.

4.3.6 Dynamic behaviour and Distributed functionality

The middleware architecture offers a method of changing or extending the behaviour and the visualization of an application runtime. An application could be extended when the user needs additional functionality. The extension mechanism uses signals in the same way that messages are transmitted in the architecture. The added functionality could be attached to the application and loaded when needed or it could be passed with a signal. The functionality could be distributed from a remote location to where the presenter and the GUI are actually loaded. See the implementation discussion for details about how this works; 4.6.4 - Dynamic behaviour.

Figure 4-18 shows how the view and the behaviour can be added at runtime. The second view has behaviour not found when the application where started. In the stopwatch example described below, 4.4 - Example of use, new presenters are added by using a dropdown box which submits create signals.

This is just an example of how to use dynamic adding of presenters. The choosing of the presenter could be a search to a remote presenter with another locality. The only thing the new presenter has to know is the extension points which will be used.

This gives the middleware many interesting possibilities. Applications can be more dynamic in respect of functionality. They can be changed with the needs of the user or be updated automatically when new versions are available.

One example where the dynamic behaviour could be used is as a widget engine like the engines in Mac OS X, Vista and in applications like Opera and Yahoo! Widget Engine. Widgets could be automatically updated when newer versions is published. New widgets could be created when functionality is used in other widgets etc.

Another scenario is a traditional client server system where an administrator of the system could add parts to the application of every user, runtime. Or change some parts to a group of users. No distribution of installation files or restarting of the application or computer is needed. Presenters could be removed runtime as well. If a user looses rights to an application, the affected parts could be removed by submitting of a signal by an administrator.

4.4 Example of use

Here we will look at an example application implemented using the middleware architecture. It is stop watch applications with the standard functionality a stop watch have. In addition the application can submit the result to a result view located in another part of the distributed application.

4.4.1 Structure

This is the overall structure of the application with to presenters available for user actions:

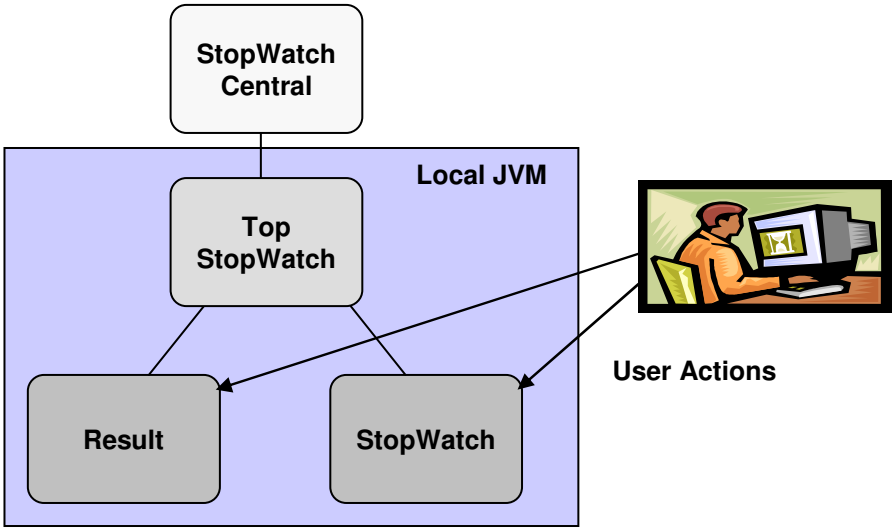

Figure 4-19: Structure of the stop watch application

Figure 4-19 shows that the stop watch consists of a central presenter, in a separate Java Virtual Machine, which bounces signals to its children. The central part of the stop watch starts up with no children connected. Children connect when they start up. The TopStopWatch presenter owns the GUI part of the application in the local JVM. The presenter offers view extensions for the children to use. The extensions provided are a composite structure and a tab folder as shown in Figure 4-20 below.

The Result presenter shows the result received from the TopStopWatch presenter. The result is shown in a row in a table for each stop watch submitting the result. When a stop watch is submitting new result the row in the result table is updated.

The StopWatch presenter is the actual stop watch. There is functionality for starting, stopping, clearing etc.

4.4.2 The view

The view of the stop watch consists of the extensions, composite and tab folder, and a status bar. The tab folder is empty at start up. (A dropdown is added to show dynamic adding of presenters for illustration purpose)

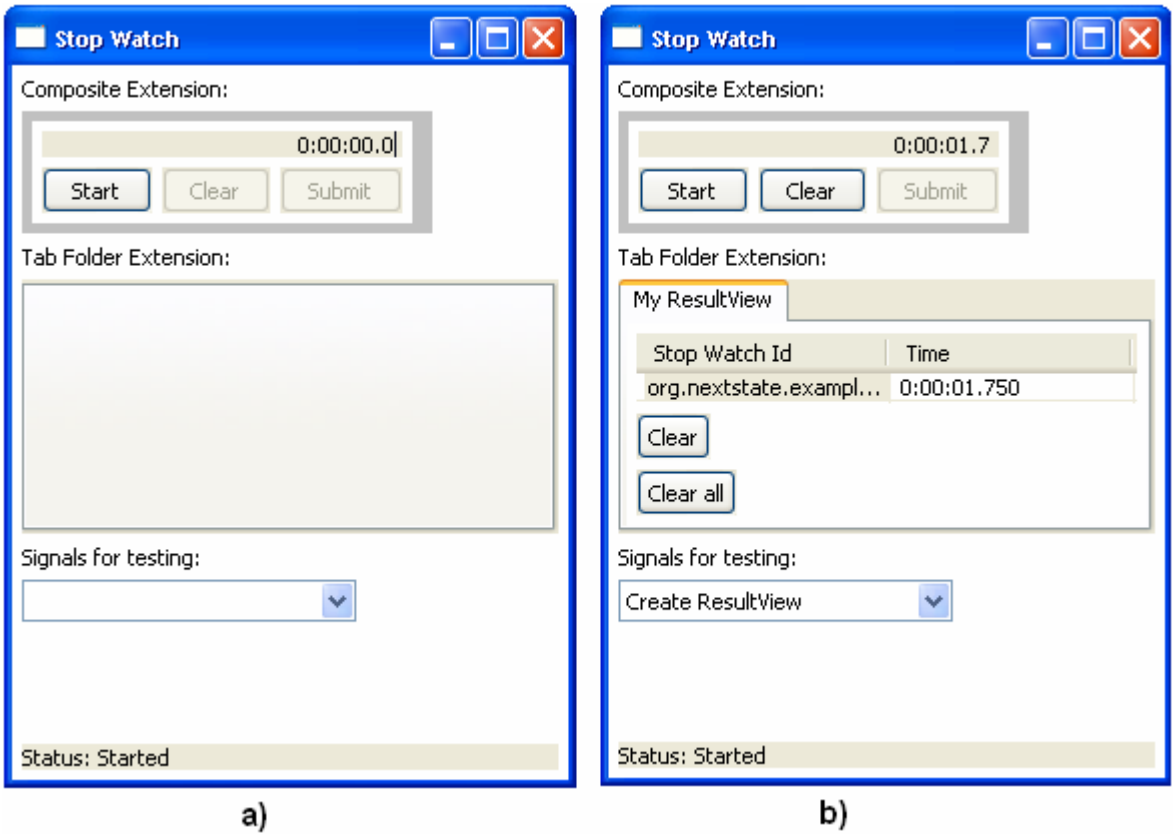

Figure 4-20: The stopwatch view

The first view in Figure 4-20 shows the stop watch at start up. A StopWatch presenter is added to the composite extension and no presenter has yet been added to the tab folder. In the second view a Result presenter has been added to the tab folder. The stop watch has submitted a result and the result is shown in the result view.

4.4.3 Controlling the view

The controlling of the view has been described partly in the section 4.3.3 - Controlling the interaction interface by state machines.

StopWatch presenter

The stopwatch controller is triggered by many different signals. Most of the signals are handled by the stopwatch presenter itself. Table 4-2 lists all signals in the stopwatch application.

Table 4-2: Signals in the stopwatch application.

Signal	Sent by (presenter)	Triggered in (presenter)	Result of the signal.
Clear	Stopwatch	Stopwatch	Clear the time in the watch.
Resume	Stopwatch	Stopwatch	Resumes the counting.
Split	Stopwatch	Stopwatch	Stop the updating of the view but continues to count.
Start	Stopwatch	Stopwatch	Start the counting.
Stop	Stopwatch	Stopwatch	Stop the counting.
Submit	Stopwatch	Result	Submits the time to the result presenter.
Suspend	Stopwatch	Stopwatch	Suspends the counting.
Unsplit	Stopwatch	Stopwatch	Continue to show the counting in the view.

All of the user events in the stopwatch view trigger a transition in the stopwatch controller. (See 4.3.3, Controlling the interaction interface by state machines, for more information on the stopwatch controller)

Result Presenter

The result presenter simply shows the result in the view when a result signal is received. The results can be cleared by pressing the clear button (see Figure 4-20). The clear all button submits a signal that clears all result views that exists in the application independent of the JVM they are running in. If a result view in another JVM is connected to the same StopWatch central presenter, Figure 4-19, this result view is cleared.

4.4.4 Dynamic behaviour

At start up the stopwatch has an empty tab folder where presenters can add there view. No result view is showing when the application is starting. When “Create ResultView” is chosen from the dropdown shown in Figure 4-20 a create signal is sent to the top presenter in the local JVM, Figure 4-19. The top presenter creates the results presenter, controller and the view and adds the view to the tab folder as shown in the second view of Figure 4-20.

The dropdown also has a choice for sending a signal that creates a stop watch in the tab folder. Then two stop watches runs independent in the application.

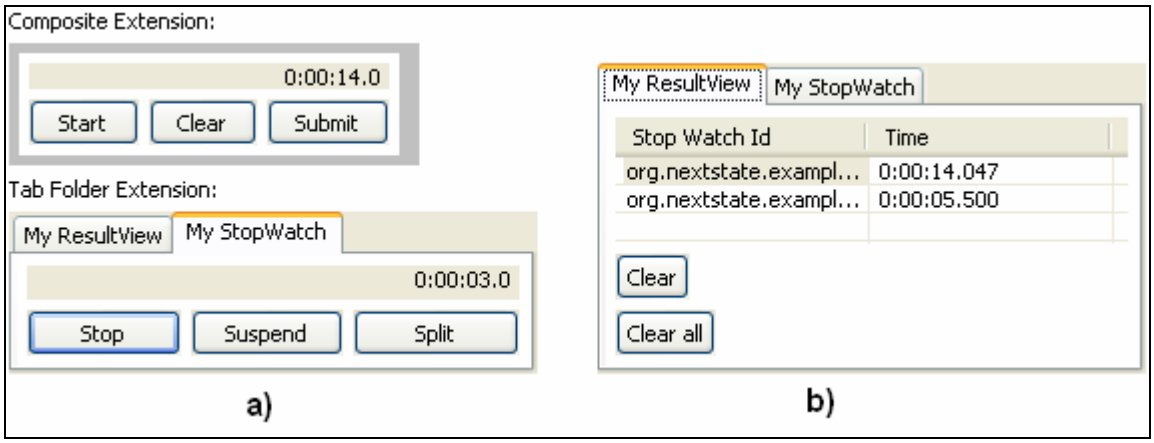

Figure 4-21: Two stopwatches

The first view in Figure 4-21 shows the two stopwatches where the first is stopped and the second is running. The second view shows the result view when both stop watches have submitted their result.

4.4.5 Filtering of signals

The Result presenter, as described in 4.4.3, is cleared when a clear signal is received independent of which part of the application that submits the signal. It might be an undesirable behaviour; that the result is cleared when someone else submits this signal. Therefore another result presenter is created with a filter on the incoming mediator from the stopwatch central presenter. This filter prevents the clear result signal from passing. This result view can therefore only be cleared by the clear button on the view itself.

The filter is implemented using the Filter interface shown in Figure 4-17: Mediator and Filter. (The method, *isAllowed(),* returns false if the signal is a clear result signal)

4.5 Principles of using the architecture

Architectures, as patterns, have situations in which they are favourable to use. The architecture described in this thesis has interesting and useful features not found in common MVC architectures (see 2.3 - Common Patterns in User Interaction) but might not be advantageous in all situations. In small and simple application, without need for dynamic changing or adding of features, the use of the architecture might be unnecessary. A single developer can keep track of the dependencies and the state which is built into the application. But in application where the internal state of the components is complex and is influenced by many other parts, a state machine based approach is beneficial. Also when more than one part is developing an application the use of this architecture is beneficial. When the signal interaction is defined the components could be developed and tested in parallel and assembled upon completion.

4.5.1 Guidelines

In the following we present a few general rules for using the architecture effectively and a list of steps to use in the development of the application.

Refine the hierarchical structure

The total hierarchical structure of the application does not have to be in place at once. Sub hierarchies can be refined during the specification and the development of the application. When the signal interaction is specified for a sub hierarchy it can be detailed in smaller parts to manage complexity.

Define the messaging API

The signals and the sequence of the interaction is an important description of the behaviour of the application. Use class diagram to define the API for the signals and sequence diagram for the interaction. When this is defined the development of the modules are separated and can be completed with no need for stubs or mocking of other modules. During the development, a module can be tested in a container, where the signal interaction is simulated, and run in isolation without other modules in the application. Finally an integration test has to be conducted to verify the interaction between the modules.

Reuse of modules

Every application consists of subsystems. If an application does not have a subsystem, it is a subsystem by its own. This might not be true for absolutely all applications. But most of the applications are designed and built in separate modules. It is important for the architecture to support and guide a well designed modularization of the application.

The separation of the modules and the definition of the interaction for each module make modules easy to reuse and integrate in other settings and applications. Presenters with similar functionality but different look can use the same presenter and controller and just vary the view implementation. In an experiment (not described in the thesis) the same presenter and controller have been used with one view located in a composite GUI element and another view as a menu. Here only the extension point that the views were using where different. The top presenters are often useful to reuse when developing applications. They already have the basic functionality implemented. Also composite states with transitions needed in many controllers are useful to reuse.

Extension points in the application

When designing a dynamic application it is important to have a set with well defined extension points. These points show where other parts can offer contribution to the application in form of extensions. A useful extension point has information about where, in

the visual view, the extension is located. The signals that might be sent to the contribution have to be described as well as which signals creates and removes the extension.

Controlling the flow of signals

All signals are broadcasted throughout the hierarchy of presenters. Some signals might just be of interest for a part of the hierarchy and does not need to be forwarded to every presenter. For example, a component which is a sub hierarchy inside a larger hierarchy does not need to submit every signal to the parent. Only signals that have interests for the parent are forwarded upwards, the others can be suppressed.

There could also be a considerable amount of signals passing. In an application running in a single machine there are no problems with handling the signals but in a distributed environment there might be an overhead which might cause trouble.

To control the flow of signals a filter is added to the port pointing to the parent of the component (see 4.3.4 - Filtering of signals). The filter has the rules for suppressing the component internal signals.

4.5.2 Most useful areas for the middleware

Any user interaction application can be developed by using the middleware. But in some areas the architecture is particularly usefully:

- The middleware makes the user-computer interaction more flexible. Applications can receive input from multiple sources concurrently which affect the application logic and update the user interface. Alternative input can be used in different situation and for different users with varying needs. Multiple user interfaces for viewing can be added at runtime as well as the sources from where inputs are received. An example could be an application for monitoring patients. Sensors on the patients are the input and the information is both displayed on a standalone application and submitted to mobile phones.
- Controlling complex user interface. State machines are a powerful and effective tool to use for controlling state of applications. UML is a standard notation with a visualisation for the state machines structure. This makes state machines very useful when designing and developing complex user interaction applications.
- Applications with need for responsive interface and asynchronous update, from users and the system. The middleware should be used in applications with need for constant user input without being unresponsive.
- Applications with need for dynamic updates and changes where not all areas of the functionality is known and is needed to be updated.
- Architectures using other devices than keyboard-mouse for interaction. Since the middleware has a clear separation between the view, which is the interaction

interface, and the controller, where the logic is found, it is ideal for adapting new interaction devices.

4.5.3 Pitfalls

There are some pitfalls developers using the middleware should be aware of. One of the changes the developer has to take into account is the change of computation model. As the middleware uses asynchronous messaging it makes it difficult to use the usual call stack for following the interaction [Hohpe '06]. This makes debugging of applications more difficult and a tool for tracing state changes of the controllers should be used.

The protocol for the signal interaction between components in the application must be carefully designed otherwise signals might be forwarded in infinite loops causing the application to stall.

4.5.4 Process of development

Here are some steps to follow when developing an application using this architecture. This is experience gathered from using the architecture while developing the example applications. It is not a complete methodology, just a start and hint of a process which might be useful. Further analysis is needed to add valuable experience and gain a better insight of usage.

Steps for the entirety application:

- Define the hierarchical structure of the application. Design the overall structure and describe what features go with each presenter. It does not have to be the tiniest details; presenters can be split in smaller parts later in the process.
- Define the signals and the interaction of the presenters in the structure. It is important to define the contract to which the communication is based on.

Steps for each presenter:

- Define the states and the transitions of the controller and what actions should be executed on a transition.
- Define the view interface. The listener which can be attached and the getters and the setters for the elements in the view.
- Define the actions and their usage of the view interface.

The steps for the presenter do not need to be incremental but should be iterative. If a presenter turns out to be too large to manage, it can be split in smaller parts. To get a head start in the development the central and the local top presenters should be reused.

4.6 Implementation of the middleware

In the following there is a description of the implementation of the architecture.

The middleware is implemented in Java but any programming language can be used for the implementation. The programming language used needs to have functionality for user interaction, multithreading, as well as features for dynamic loading and reflection. In addition to the Java language and the Java API, Standard Widget Toolkit [SWT '07] were used for user interaction as the GUI library (see 2.4.2 - Standard Widget Toolkit).

The architecture has a core containing a state machine engine which executes the controllers in an application and a library of supporting classes for creation of the parts needed to execute state machines (See Figure 4-3: Layers in the architecture). There is a library with classes which is specialized for using the presentation architecture and act as a base for building the applications.

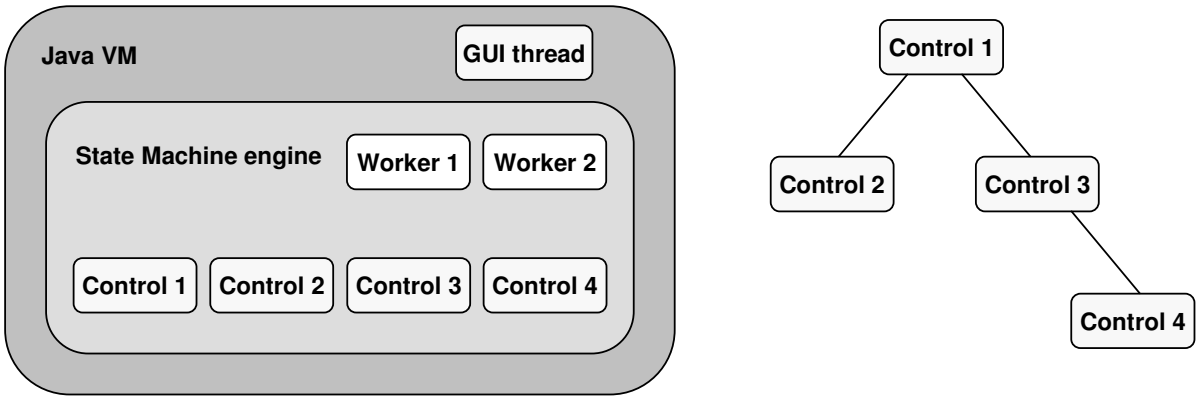

Figure 4-22: The runtime of the state machine architecture and the hierarchy of controllers[1]

The runtime environment is a state machine engine running inside a Java Virtual Machine (JVM). Figure 4-22 shows the engine with the contained controllers and also the actual hierarchical structure of the controllers. The controllers are executed independently of how they are organized in the hierarchy. The workers of the engine execute the controllers in turn to process all signals arriving to the controllers.

The JVM also runs the thread for interaction with the graphical user interface (the Event Dispatch Thread). Only this thread is allowed to update the GUI, other threads accessing the GUI components result in exceptions. (See 2.4.2 - Standard Widget Toolkit)

The GUI library used in the implementation of the middleware is the Standard Widget Toolkit from Eclipse [SWT '07]. The implementation is not locked to SWT. Swing [Walrath '04] or some other GUI library could be added without much effort.

[1] It is the presenters that are organized in a hierarchy. The figure is simplified by showing the controller in the hierarchy to match the names shown in the State Machine engine. There is a one-to-one relation between the presenter and the controller.

4.6.1 Broadcast bounce

In this implementation of the middleware it is the controller that does the actual forwarding of the signals through the mediators in the hierarchy. The controller has two outgoing mediators: one to the parent and one which broadcast the signal to all children. (See Figure 4-6: The presenter - controller hierarchy) The bouncing behaviour is achieved by a composite state in the controller which forwards the signals in the hierarchy. The composite state has self-transitions for forwarding signals either to the parent or to the children. The choice is made based on the property the signal has. (A special hierarchy signal type is defined for passing through the hierarchy that has a property for the direction the signal should be forwarded in)

To bounce the signal to the top of the hierarchy the top presenter uses another transition which forwards the signal downwards in the hierarchy again.

There is also a property the hierarchy signal could use to forward signals to the siblings of the presenter. This is accomplished by another transition in the composite state.

Another way of implementing the broadcast bounce could be to let the signals pass the controller without being executed by the controller on the way to the top, only do the execution on the way back. But this would stop the ability for having local top presenter that bounces signals locally in a smaller part of an application.

4.6.2 State machine execution

The implementation of the state machine uses much of the same terminology which is used in UML state machines with actions, guards, transitions and active state configuration [Booch et al. '05]. State machines, states and transitions are all implemented as Java classes. Actions and guards are Java interfaces which can be implemented as usual classes or as anonymous inner classes in the state machine class. If an action is implemented as a standalone class instead of an anonymous inner class, it can be reused by other controllers.

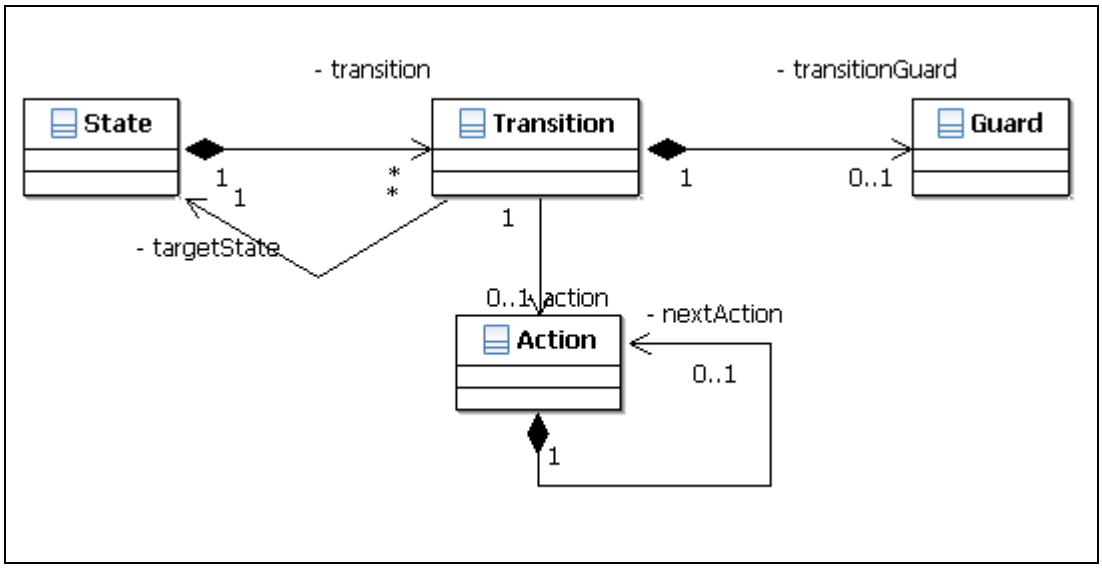

Figure 4-23: The state transition concept model

Figure 4-23 shows the connections between the elements used in the implementation of the UML state machine. A state has a collection of transitions which is checked in case of an event. The guard of the transition is checked for validating the signal. If one of the transitions has a guard which condition are satisfied the transition is "fired". The action associated with the transition is executed and the target state is entered and stored in the active state configuration.

There is no guarantee in which order the transitions is checked. The first transition which satisfies the guard conditions is fired.

The middleware has a composite state implementation which can hold other states. The composite state has a reference to the initial state in the composite.

Not all of the UML 2.0 specification has been implemented in the middleware. The UML activity workflow with branching, merging and forking etc. has not been implemented. Actions are instead implemented using the Chain of Responsibility pattern [Gamma '94] as a "lightweight" workflow to have the ability to execute more than one action in one transition.

4.6.3 The View library

The view library layer consists of a collection of classes and interfaces that have common features needed for applications in the presentation middleware. The classes contain common functionality and are used by the applications by inheritance. The view library builds upon the state machine runtime implementation in combination with a GUI library. This adds GUI functionality in addition to the state machine engine implementation which consists of the functionality for state machine execution, actions, transitions etc.

An application, which is using this architecture, extends the abstract classes in the view library listed in Table 4-3. This gives the basic functionality for the parts that are developed and gives a pattern to follow in the implementation. This eases the development of state machine based applications.

Table 4-3: Main view library classes

Superclass	Description
PresenterImpl	This is the common presenter and is used for presenters that are children of another presenter.
MainPresenterImpl	The main presenter initializes and starts the thread engine before activating the GUI. This could be the top presenter in a system.
RemotePresenterImpl	The remote presenter connects to a remote presenter and/or opens a port for remote children to connect to. This could be the top presenter in a system.
ControllerImpl	The controller is the specialized state machine with mediators both to the parent and to the children. The superclass for all controllers.
ViewImpl	The view super class have hocks for showing and closing the view. The common superclass for all views.
ViewAction	This action has to be subclassed when a change in the user interface is performed.
GuardImpl	A basic guard to be subclassed for adding the guard logic.

View extensions

A view extension, which is used as a slot for other parts to present their view elements, is simply a map with the extensions owned by the view. This is the only way a child presenter can access the actual GUI. The key in the map is the name that is given to the extension. The actual view extension consists of a name, an extension type (listed in Table 4-4) and the user interface object from the GUI library which is used for publishing the view elements.[1]

Table 4-4: View Extensions

Extension	Description
Composite	A defined area of the GUI.
Menu	The menu bar on the main window in the application

[1] The view extension is closely connected to the GUI library used for the implementation since the extension contains an object from the GUI library.

Popup menu	A sub menu that already might have elements attached.
Shell	The whole window.
Status bar	The status bar at the bottom of the window.
Tab	A tab folder where separate tabs can be added.

A composite extension is a defined area in the GUI which a view offers for extension. It can be a whole window or a just a small part of the screen

4.6.4 Dynamic behaviour

The middleware has the ability for dynamic adding and removing presenters. The presenters could be located on the classpath or be passed from a presenter in another JVM. This is implemented by using the reflection mechanism in Java. This is achieved by sending a specialized signal which contains the information needed to load the classes. The information needed is class name, location of the class and which view extension to use.

If the presenter, the view and the controller is not known by the running JVM the definitions has to be passed with the creation signal as well. Any kind of data can be transferred by a signal. Even presenters and state machines can be transferred with a signal. For this purpose a specialized two-phase class loader has been built that loads presenters in two steps. In the first phase the remote part loads the presenter into memory. The remote part adds the class definition to the create signal. In the second phase the signal is received by the presenter that is going to create the new presenter. This presenter loads the new presenter by using the class definition passed by the signal.

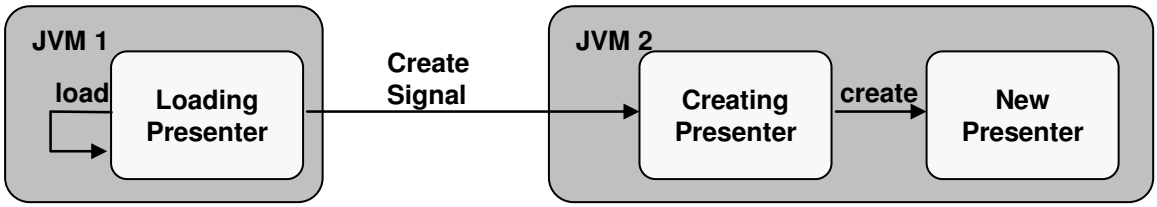

Figure 4-24: Two-phase class loading

Figure 4-24 shows a presenter loading another presenter from the classpath in JVM 1. The loaded presenter is added to a create signal and forwarded to the creating presenter in JVM 2. The create presenter loads and creates the new presenter from the class definition passed by the signal. The new presenter does not exist on the classpath in the JVM 2.

4.6.5 The State Machine Runtime Engine

The runtime environment of the middleware is a state machine engine that runs on top of the Java Virtual Machine. The engine has workers which performs the actual execution of the state machines. An engine can use one or many workers for the execution depending on the need and performance of the application.

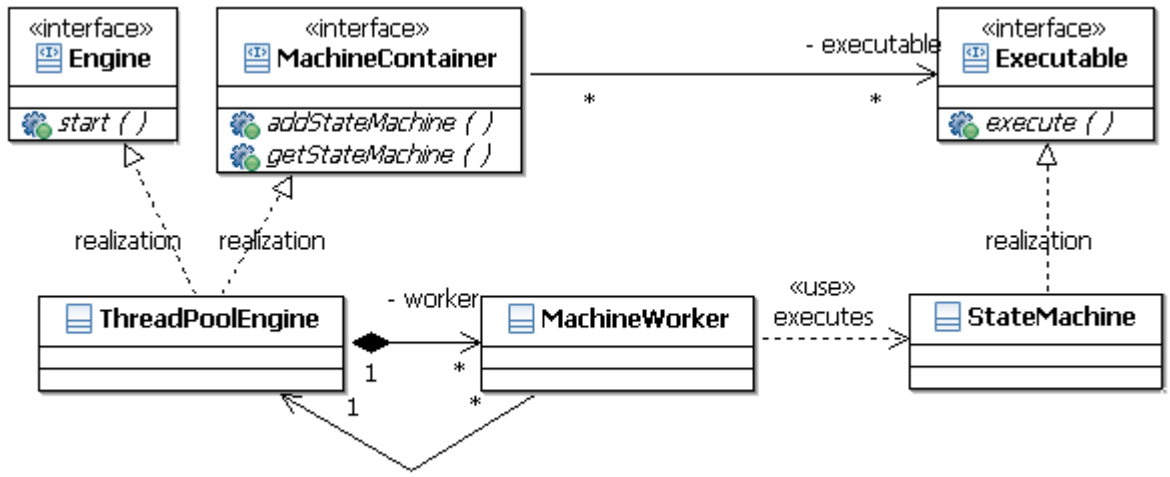

Figure 4-25: The engine concept model

The engine (ThreadPoolEngine), in Figure 4-25 is an implementation of the Engine and MachineContainer interfaces. The Engine interface controls the starting and stopping of the engine. The MachineContainer holds the executable that the engine is executing. An implementation of an Executable is an object that the MachineWorker can execute. The MachineWorker does the actual execution of the state machine. The worker gets state machines from the engine and executes the incoming signals the state machine might have stored. The engine contains one or more workers.

One of the engines implemented uses the Concurrent Utilities API in Java SE 5.0. The ThreadPoolExecutor and the ExecutorService is used for handling the threads used for executing the machine workers in the engine. Another engine implementation uses plain Java tread handling.

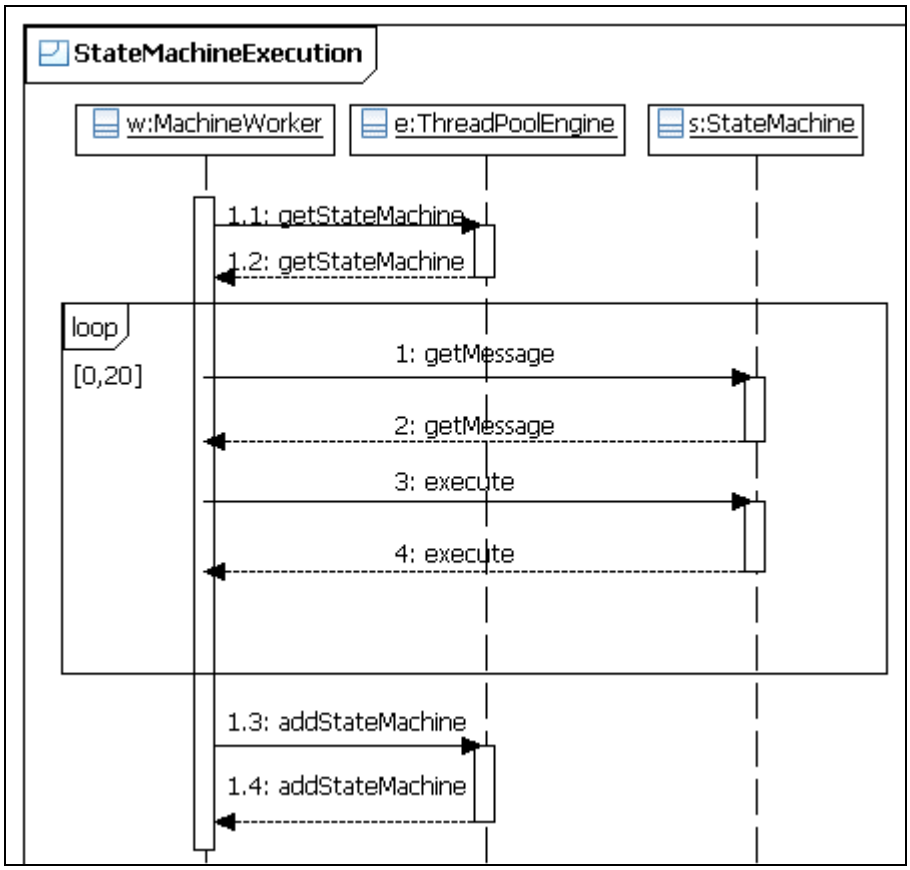

Figure 4-26: A worker executing a state machine

Figure 4-26 is an example how the ThreadPoolEngine is implemented. The worker gets a state machine for execution from the engine and executes the signal on the state machine. The MachineWorker continues until there are no more signals or until the configured maximum numbers of signals for execution limit is reached. Finally the state machine is returned to the engine.

There are a number of possible configurations that could be optimized for performance, including: number of concurrent threads running the workers, number of signals for execution before a worker continues with the next state machine etc.

Thread handling

In the implementation of the middleware the workers run one thread each. In addition to the threads in the engine there is one thread which runs the GUI (Event Dispatch Thread), see 2.4.2 - Standard Widget Toolkit. The Event Dispatch Thread executes all GUI rendering activities including the event handling in the application. This thread only forward the signals created in the listeners and do not do any state machine execution. In the same way the threads owned by the engine is not performing any GUI rendering. View actions are used to separate these tasks, see 4.3.2 - Execution of View Actions.

4.6.6 Model Driven Architecture

Model Driven Development (MDA) is an initiative from the Object Management Group [OMG '07] to separate the business and application logic from the underlying platform technology. The aim of MDA is to shift focus from coding to modelling. MDA has a more abstract focus where the model transformation takes care of the platform dependent code generation [Raistrick '04].

Code generation are beneficially in many ways. Code generation from models can:

- Adapt code to different platforms
- Models are easier to document and understand (visual)
- Analyse the model formally
- Dramatic reduction of errors [Bræk '00]

Model Driven approach

Model Driven Development (MDD) has promised productivity, quality and platform independence but has not been as successfully as hoped. There is a potential for a seamless link between the model and the implementation of an application but the benefit has so far been limited in the software industry.

An application which is going to be implemented with the middleware, presented in this thesis, is best defined in a UML model. UML has the possibility for expressing the connections between the presenters through the ports in the hierarchy and has the state machine diagram for describing the state transitions the controller can execute. The signals are defined in a class diagram. Sequence and Activity diagram could be used to support the design process.

The architecture of the middleware has assets which fit well in to a code generation approach:

- It is a well formed and strict structure with a clear separation of concerns. The different parts are implemented using the triple of the presenter – controller – view. The properties of the controller, actions, guards, could be implemented as anonymous inner classes or concrete classes.
- Modular structure - parts of the application can be modelled and implemented/ generated separately and then assembled.

To use the middleware in a MDD approach a UML profile should be defined. The architectural parts of the middleware, presenter, view, action etc, should be stereotypes used to express the structure and the behaviour of an application.

Experiences from code generation

With the middleware architecture definition in place a code generator should be possible to construct. The transformation from model to code should under most circumstances be easy to perform.

Some limited experiments with code generation from models have been performed during the thesis work. Different code generating tools have been used. IBM Rational Modeller v. 6.0 (xTools) [Swithinbank '05], openArchitectureWare v. 4.1 [oAW] and Acceleo v. 1.1 [Acceleo] has been evaluated.

Some of the experiences from the usage of the tools:

- Good knowledge of the UML meta-model is needed – To construct a code generator extensive knowledge of the meta-model is needed, not only to understand the model language itself but to understand how to construct the code generation transformations.
- Knowledge of the UML meta-model implementation. To be able to navigate and debug the code generation process it is important to know how the meta-model is implemented and how to use it. The Eclipse UML2 meta-model [Budinsky et al. '03] is used for describing models and is an implementation of the UML definition. In the process of defining code templates for generating the code it is crucial to recognize the meta-model elements the templates have to navigate and try to use.
- Since the templates, for the transformation definitions, uses the complex UML meta-model the tools which have code completion and syntax highlighting for the code generation templates is much easier to use.
- The time of the cycles when developing the code generator is essential. IBM Rational Modeller has a much long cycles compared to Acceleo's nearly instant preview functionality.
- Ability for customization of the generation. All tools evaluated had the ability for customization by using Java.

5. Re-implementation of the analysed application

The analysis of the CM application, 3 - Analysis of a Case Management application, revealed some functional drawbacks due to the architecture. The state management of the modules, notification between modules, navigation and long running process were some of the problems the application ran into.

We have re-implemented parts of the application with applying our special middleware. The implementation is just an example of how the middleware could solve the problems found and is an illustration of the functionality that the architecture gives.

This example is a simple registration application where a user can register information and submit to a persistent store. The user has to be logged in to be able to submit the registration.

The application consists of presenters in different Java Virtual Machines (JVM).

5.1 Structure overview

The example consists of the following presenters where the CentralRegistration and the RegistrationService are presenters in the central part of the application. The others are part of a local JVM. (The CentralRegistration can have any number of TopRegistration presenters connected)

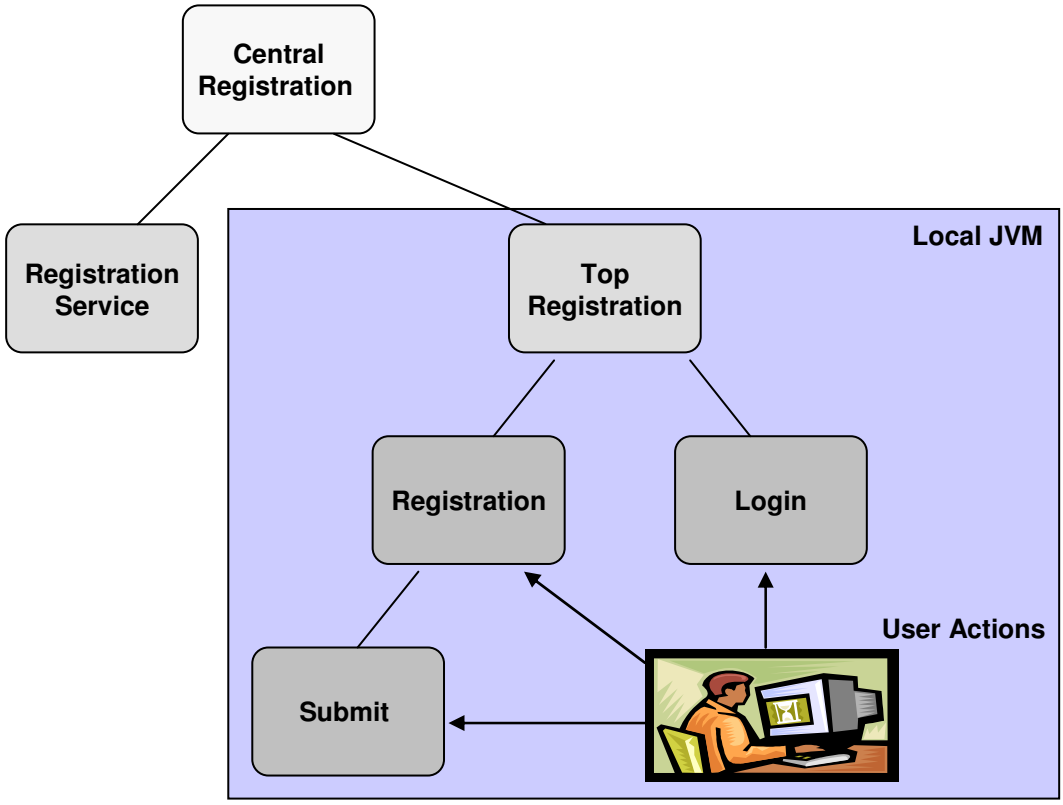

Figure 5-1: Presenter structure of the example

Figure 5-1 shows that three presenters are available for user actions. When the user interacts with one of them a signal is created and submitted to the controller of the presenter. All presenters used in the example are listed in Table 5-1.

Table 5-1: Presenters in the example

Presenter	**Description**
CentralRegistration	The top presenter of the application only bounces the signals received from the children. This presenter can be connected to from children in other JVM's which use this presenter as their top presenter. This presenter also starts the state machine engine.
RegistrationService	This presenter is a child to the CentralRegistrationPresenter which is created when the application starts up. The presenter receives Registration signals that are persisted in the application database. Login signals are also handled.
TopRegistration	This is the top presenter in the local JVM which shows the GUI to the user. The presenter connects to the central presenter at start up and forwards the signals from the children. This presenter owns the GUI and starts the state machine engine in this JVM.
Registration	Holds the registration GUI and the state management for the registration and preview of the information entered.
Submit	This presenter controls the status of the submit button for the registration. The user has to be logged in to be able to submit a registration.
Login	A presenter for login of a user and account information. The user sends the username and password to login.
SearchUser	This presenter is for searching and choosing name of users for the Registration presenter.
Log	A presenter for showing all signals passing.

The SearchUser and the Log presenter is not shown in the structure overview, Figure 5-1, since they are not initialised at start up. They are created if the user asks for them. They operate on the same hierarchical level as the Registration and Login presenters; children of the TopRegistration. The SearchUser is available for user actions.

5.2 The internal structure of the presenters

Here follows a description of the internal structure of the presenters used. The transitions of the controller and the look of the view are described.

5.2.1 CentralRegistration Presenter

The central presenter is the top of the application and bounces all signals to the children.

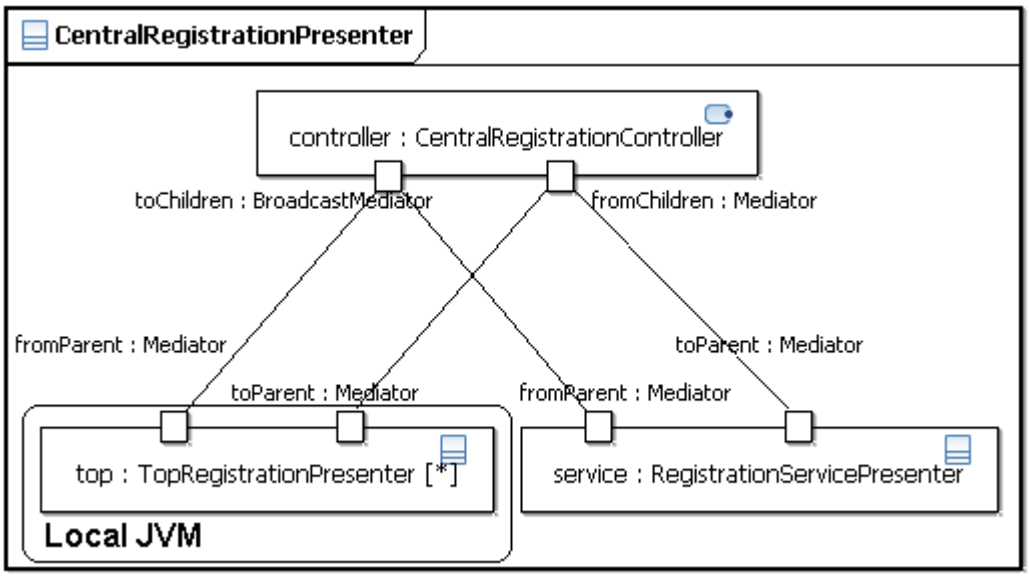

Figure 5-2: Composite structure of the Central presenter

Figure 5-2 shows the top structure of the application where local TopRegistration presenter can connect to the CentralRegistration presenter from a remote JVM. The RegistrationService presenter is created at start up and present during the lifetime of the CentralRegistration presenter.

The controller simply bounces all signals to the presenter's children. No one of the presenters in the central JVM has a view.

A filter is added to the toChildren Broadcast mediator on the CentralRegistrationController for letting signals through to only the children with the id that matches the signal. This prevents signals which are not of interest of other children to be forwarded to them. Signals without receiving id are broadcast to all children.

5.2.2 RegistrationService Presenter

This presenter illustrates a persistent storage for information in the application. This presenter store registrations and validates user login.

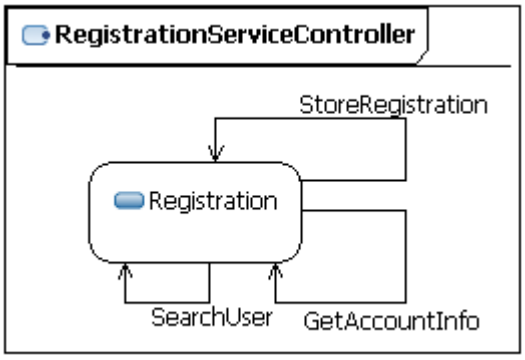

Figure 5-3: Registration service controller

As Figure 5-3 shows the only transitions the controller has, in this simple example, is to store registration, search for users and validate user information (GetAccountInfo).

5.2.3 TopRegistration Presenter

This is the local top presenter which owns the state machine engine and the GUI for the local JVM. The presenter owns the GUI and offers extensions for a tab folder, a composite structure and a menu bar that the children can use.

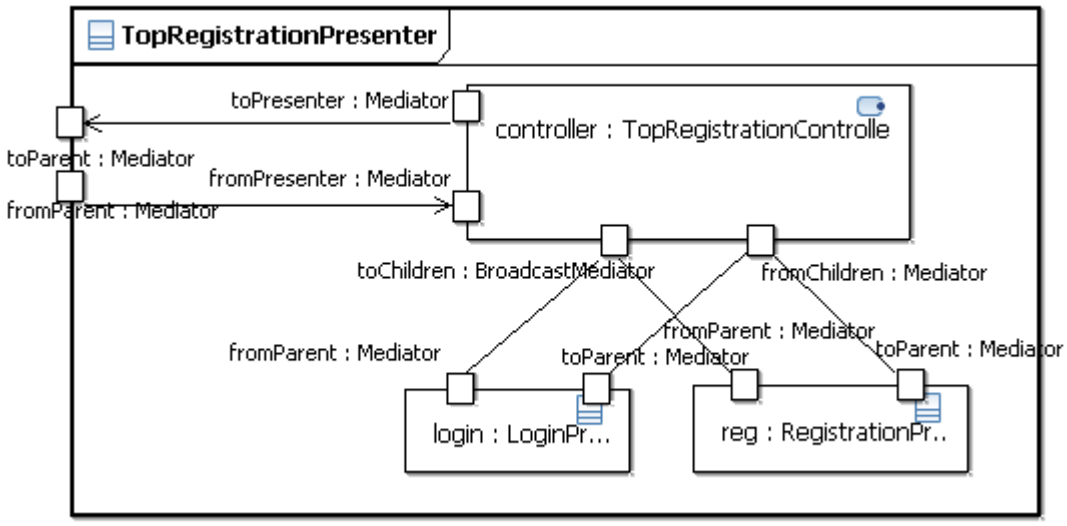

Figure 5-4: Composite structure of the Top Registration presenter

Figure 5-4 shows the structure of the local top presenter and its children. Here the registration presenter and the login presenter are connected.

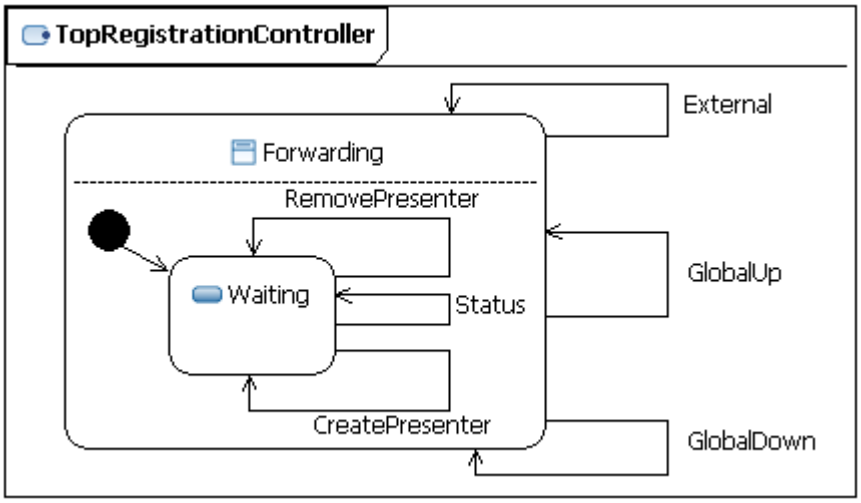

Figure 5-5: The Top Registration controller

Figure 5-5 shows the controller for the TopRegistration presenter which has a composite state with forwarding transitions. The composites state contains a single state which has transitions for creation and removal of presenters and a transition for showing text in the status bar. The status transition is triggered on status signals.

The view for the top registration is just the containing window, the extension points described above and the status bar (as shown in the bottom of Figure 5-6 below). The status bar is used for notification of state changes in other presenters.

Signals to the top presenter are ether passed to the parent or bounced depending on the type of the signal. External signals are forwarded to the remote CentralRegistration presenter others are forwarded downwards again.

5.2.4 Registration Presenter

The registration presenter is a child to the TopRegistration presenter and adds its view as a tab to the tab folder. The registration is a number of fields with information for the registration. The registration has two states for the information, edit and preview. The presenter has a composite extension where the submit registration presenter is connected to. This is shown as in the bottom right corner of the Registration tab in Figure 5-6.

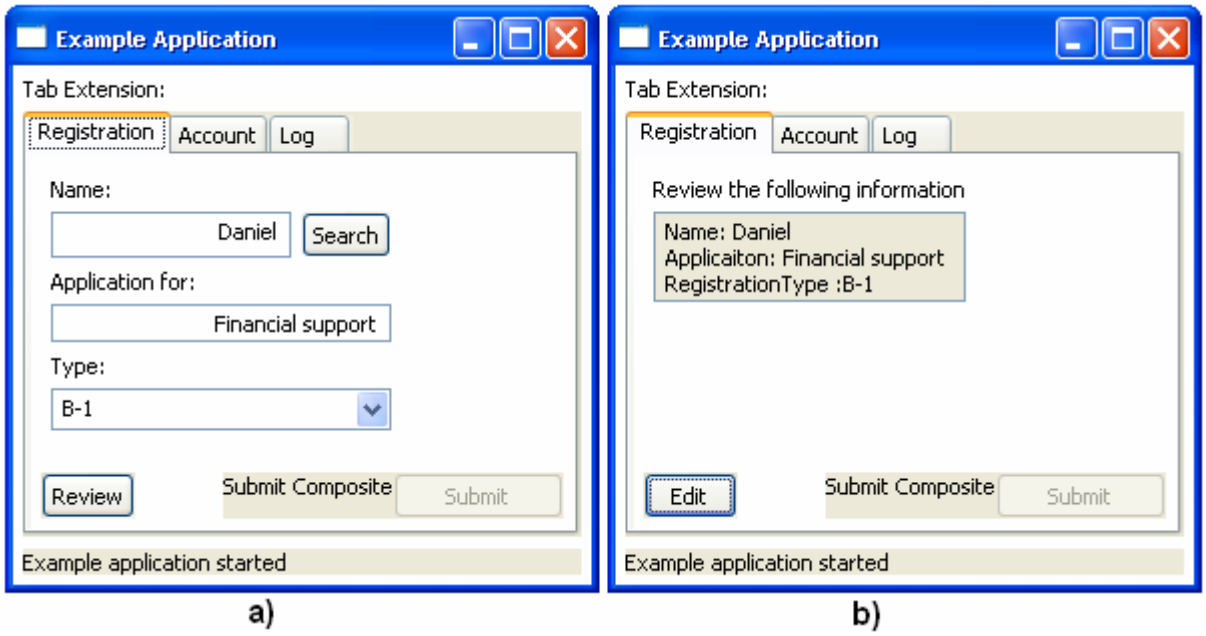

Figure 5-6: Edit and reviewing the registration

The first view in Figure 5-6, a), shows the registration of the application information. The second, b), shows the review of the information after the review button in a) has been pressed. To navigate back to a) the Edit button is pressed. If the user is logged in the Submit button will be enabled in b).

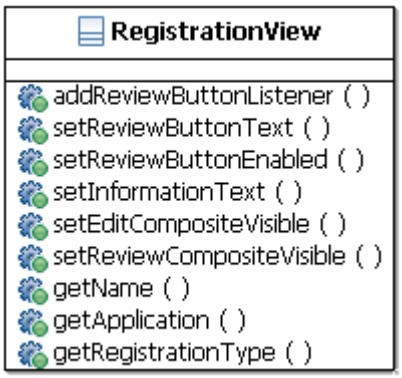

Figure 5-7: Registration view class interface

The registration controller is responsible for changing the registration view. The view implements a strict interface that the controller uses. Figure 5-7 shows the interface of the registration view. This is the methods the controller uses to alter the view.

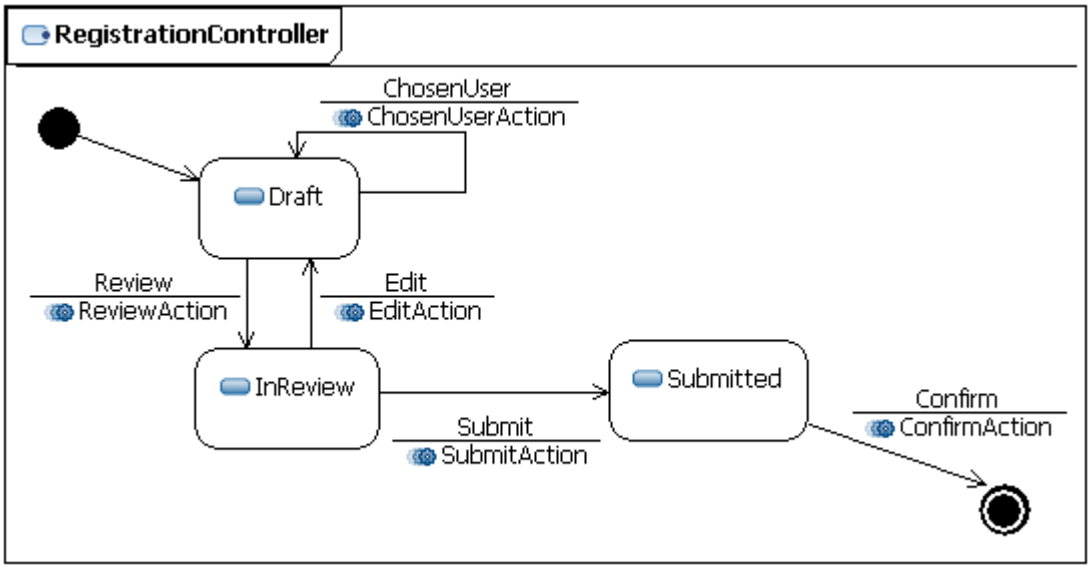

Figure 5-8: The controller of the registration presenter

Figure 5-8 shows how the controller for the registration presenter handles the state changes in the view. The Draft state is shown as the first view in Figure 5-6 (a). The InReview, Submitted and final state is shown as the second view in Figure 5-6 (b).

(In the Submitted and the final state the text in the view is changed from "Review the following information" to "Submitting…" and "Confirmed…")

Signals

The Review and Edit transitions are triggered by signals submitted by the Registration view itself. The Submit and Confirm transitions are triggered by signals received from other presenters. The Submit signal is sent by the Submit presenter and the Confirm signal is sent by the RegistrationService presenter after the registration is stored. (It is the controller in the triplet, Figure 4-4, that performs the actually forwarding of the signal)

5.2.5 Submit Presenter

This presenter controls the status of the submit button for the registration. The user has to be logged in to be able to submit a registration.

The Submit view is only the button in the bottom right corner of the Registration tab as shown in Figure 5-6.

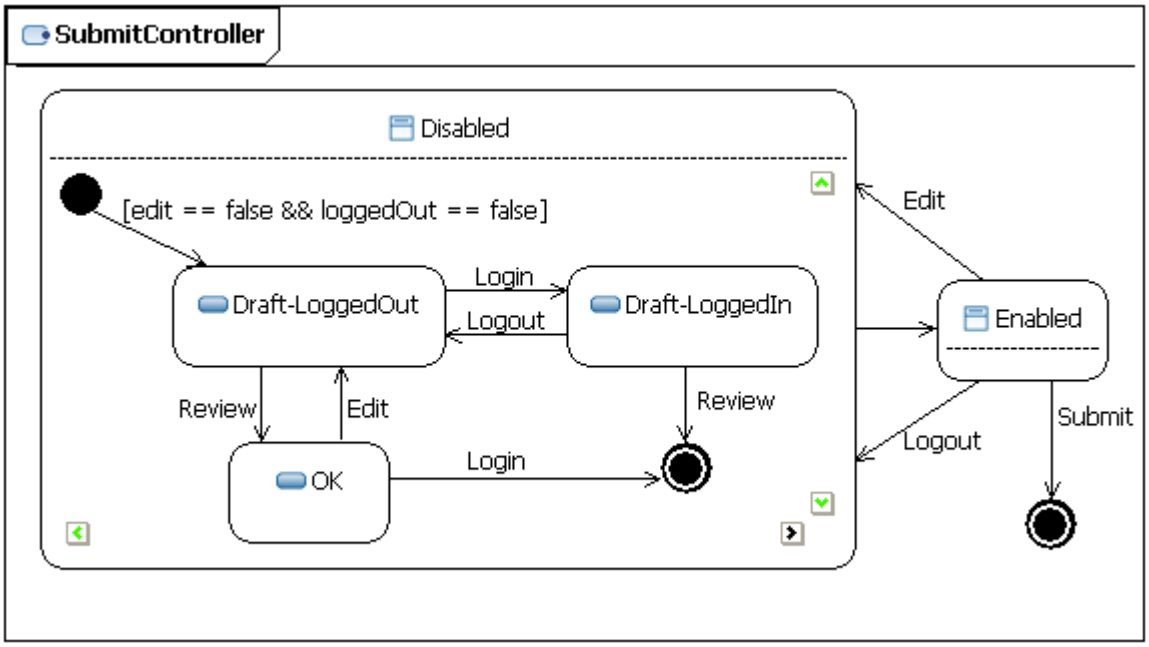

Figure 5-9: The submit controller[1]

Figure 5-9 shows the states and the transitions for the submit controller. The state Disabled and Enabled controls the submit button in the view and has actions which makes the button enabled or disabled. The internal states of the Disabled composite state handle the different variations the Disabled state can have. Dependent on if the user is logged inn or not a Review signal should lead to different actions. When the controller leaves the Enabled state either the OK or the Draft-LoggedIn is activated in the Disable composite state dependent on the transition occurred. (Two initial transitions are missing, see the footnote of Figure 5-9)

Signals

The review and the Edit transitions are triggered by signals sent from the Registration presenter. The Login and Logout transitions are triggered by signals sent by the Login presenter. Finally the Submit transition is submitted by the Submit presenter itself.

5.2.6 Login Presenter

This presenter adds functionality for logging in to the application and managing the account information.

[1] Two transitions are missing in the state machine. There should be to additional initial transitions to OK and Draft-LoggedIn states with the guards loggedOut == true and edit == true. Due to limits in the UML tool used (Rational Software Modeller v 7.0) ,which can not handle more than one initial transitions, these are not shown. But the UML reference manual, Rumbaugh, J., Booch, G. & Jacobson, I. (2004) *The unified modeling language reference manual,* Boston, Addison-Wesley., declare that a initial state can have more than one transition if all transitions “cover all possible cases” (page 393).

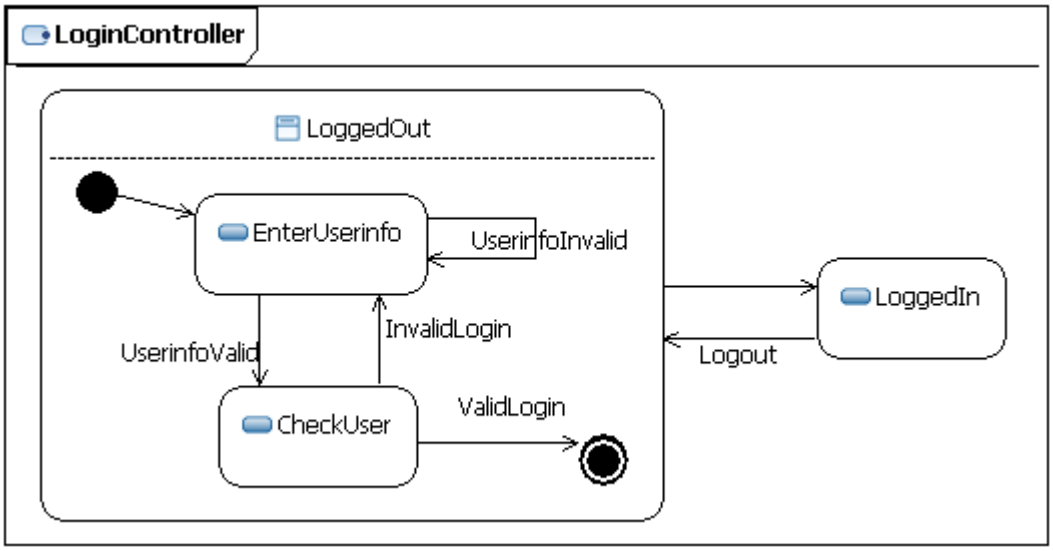

Figure 5-10: The controller for the Login presenter

As shown in Figure 5-10 the login controller has a composite state with two internal states.

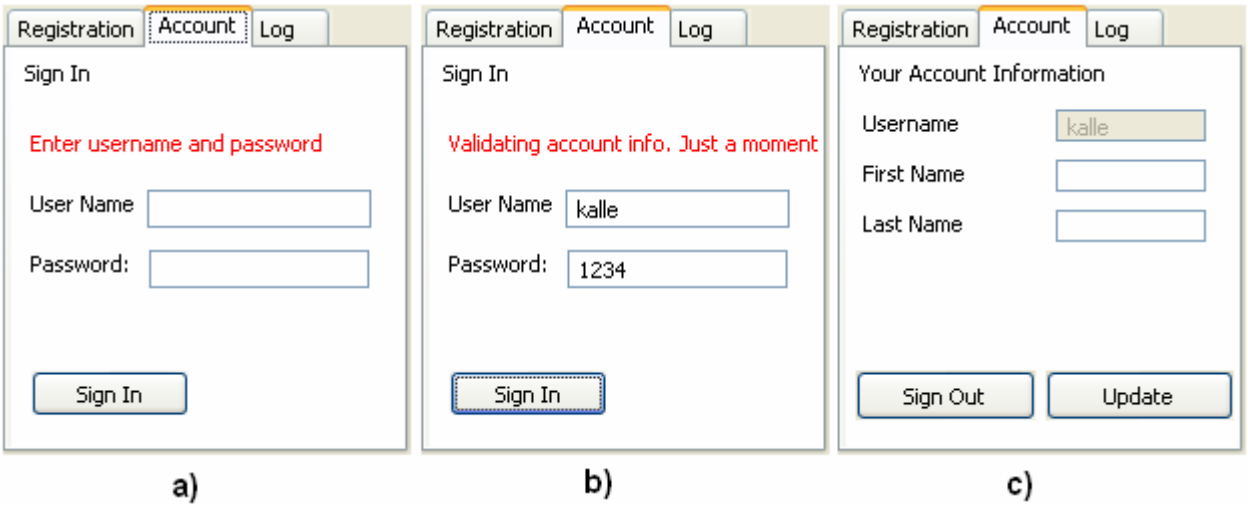

Figure 5-11: Different look of the Login view

Figure 5-11 shows the looks the login view can have. The first two (a and b) shows the view for logging in and the third (c) shows the login and account information. The first look is when the controller is in the EnterUserinfo state, the second is when the controller is in the CheckUser state and the third in LoggedIn state.

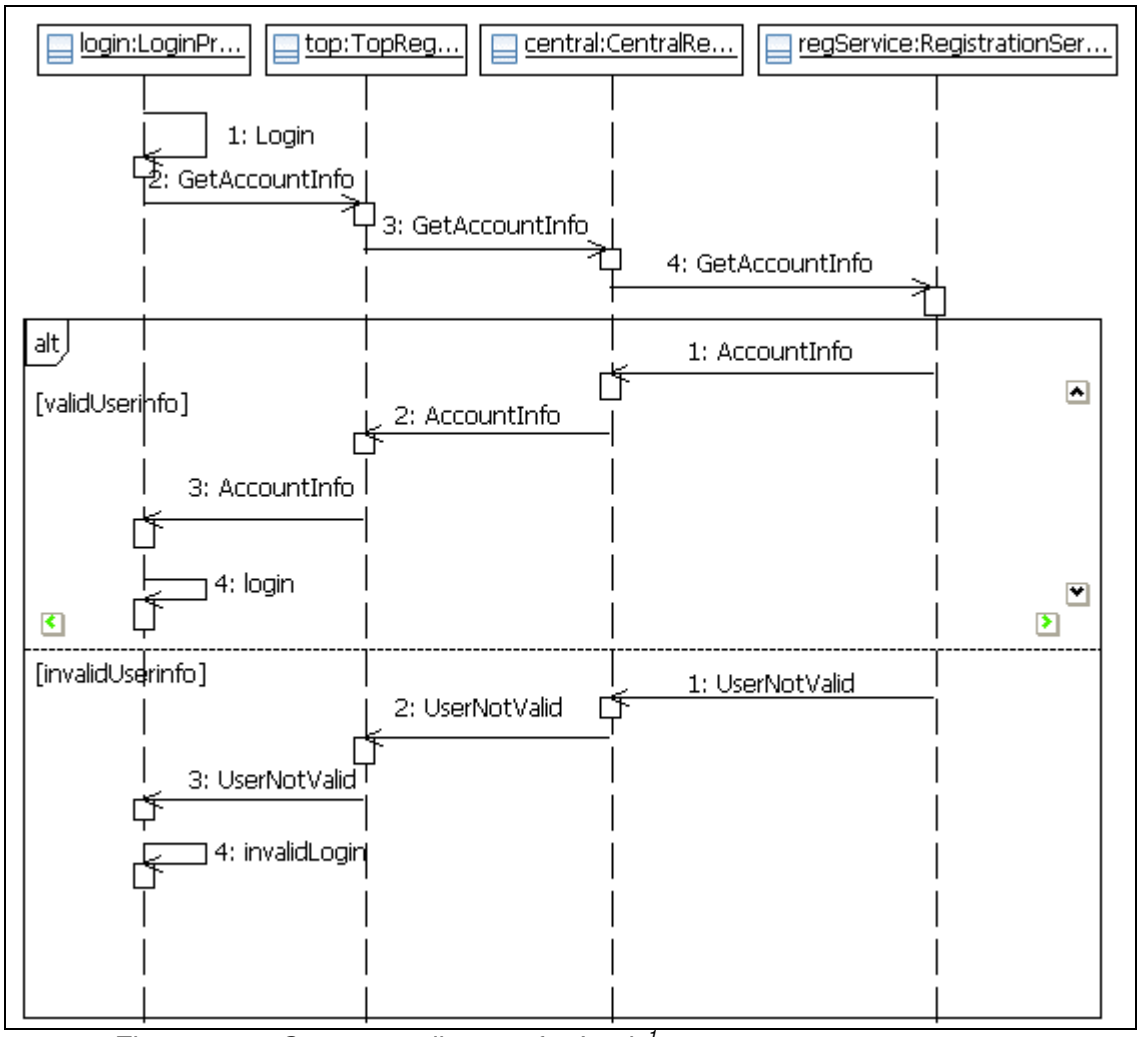

Figure 5-12: Sequence diagram for Login[1]

Figure 5-12 shows the sequence diagram for the login user action.

The login controller changes state to CheckUser when the GetAccountInfo is sent. When a signal is received the login controller changes to ether EnterUserInfo or LoggedIn state dependent on what signal arrives. In this example application the sending of the AccountInfo signal from the RegistrationService is delayed to illustrate latency in the application. The view is not changed to the LoggedIn look, as shown in Figure 5-11, before the AccountInfo signal is received.

When there are a state transition between the LoggedIn and LoggedOut state the view is changed and an information signal about the transition is forwarded. This signal triggers transition in the controller of the Submit presenter as well in the TopRegistration presenter. In the Submit presenter, Figure 5-9, different transitions can be triggered dependent of which state is active. The TopRegistration presenter has a transition for showing text in the status bar and the login and logout signals trigger this transition. This notifies the user of the change of login state even if the login view is hidden (the registration tab is active).

[1] Only the presenters are shown for simplicity. The arrows illustrate the message sent between the presenters except for 4: login and 4: invalidLogin in the alternative combined fragment that are methods on the login view.

5.2.7 SearchUser Presenter

This presenter adds search functionality for the registration presenter. In the registration view, where the name should be entered, a search can return the name instead. The Search button in the registration view, Figure 5-6, opens the SearchUser presenter which is a modal dialog window.

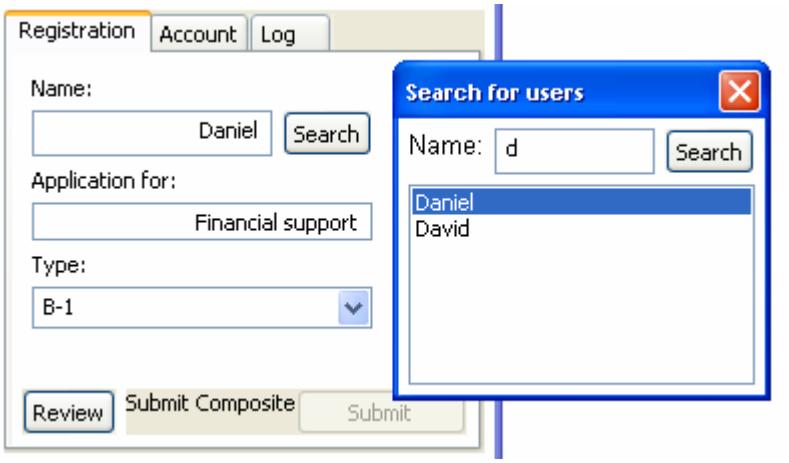

Figure 5-13: Search for user

When the search button, in the "Search for user" window (Figure 5-13), is pressed a SearchUser signal is forwarded. The signals trigger a transition in the RegistrationService, Figure 5-3, which returns a UserResult signal. This signal triggers a transition in the SearchUser controller that adds the result-set to the view. (In Figure 5-13 shown as Daniel and David) If a name in the result-set is selected the SearchUser submits a signal with the chosen name to the Registration presenter, Figure 5-8. The Registration view is updated when the signal is received; it does not have to wait until the search window is closed.

5.2.8 Log Presenter

Finally there is a presenter that logs all signals passing. This presenter is mainly for debugging purpose but could be extended to a notification view where the history of the notifications sent is shown. The log presenter is added at runtime by submitting a create signal to the Top Registration presenter. The Log presenter is added as a child to the Top presenter (in the same level as the Login and Registration presenters; see Figure 5-1). The tab extension is used for the view:

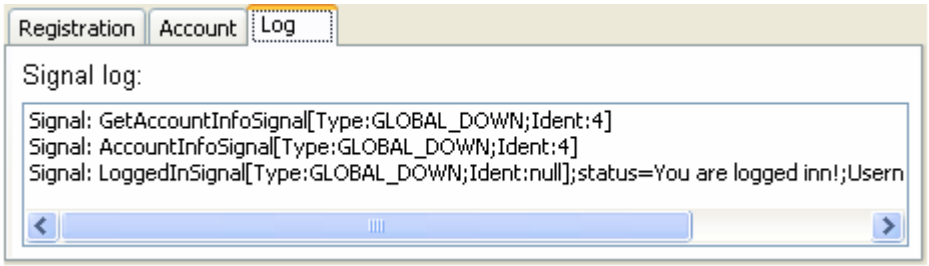

Figure 5-14: The Log view

Figure 5-14 shows the view which logs all signals the presenter receives. Since the log presenter is a child of the Top Registration presenter all signals is passed to it. To use this presenter as a history of the status with notification to the user, a filter could be added to the incoming mediator of the Log presenter. If the filter only accepts notification signals then only the notification history would be shown in the view.

5.3 Signals in the example application

Table 5-2 gives an overview over the signals that is forwarded in the application and which presenters that creates and is triggered on the different signals.

Table 5-2: Signals in the example application

Signal	**Sent by (presenter)**	**Triggered in (presenter)**	**Description**
Review	Registration	Registration, Submit	Review button has been pressed.
Edit	Registration	Registration, Submit	Edit button has been pressed.
Submit (signal)	Submit	Registration	Informs that a submit action has been performed.
SubmitRegistration	Registration	RegistrationService	Submits the registration registered in the view to the data store.
Confirm	RegistrationService	Registration	Confirmation that the registration has been stored.
LoginSignal	Login	Login	User event for login
LogoutSignal	Login	Login, Submit	User event for logout
GetAccountInfo	Login	RegistrationService	Request for validation of user information for login.
UserNotValid	RegistrationService	Login	User validation fail
AccountInfo	RegistrationService	Login	User validation OK

SearchUser	SearchUser	RegistrationService	The search for user criteria.
UserResult	RegistrationService	SearchUser	The result-set with users from the search.
ChosenUserSignal	SearchUser	Registration	A user chosen in a result-set from a search.

5.4 Testing the presenters

During development a testing environment can be used to test presenters and their behaviour on signals. The individual presenter-controller-view triplet is tested in separation with a GUI front. Signals are registered in the testing environment and can be submitted to the presenter to be tested. The testing environment is an implementation of the central classes in the view library, see 4.6.3.

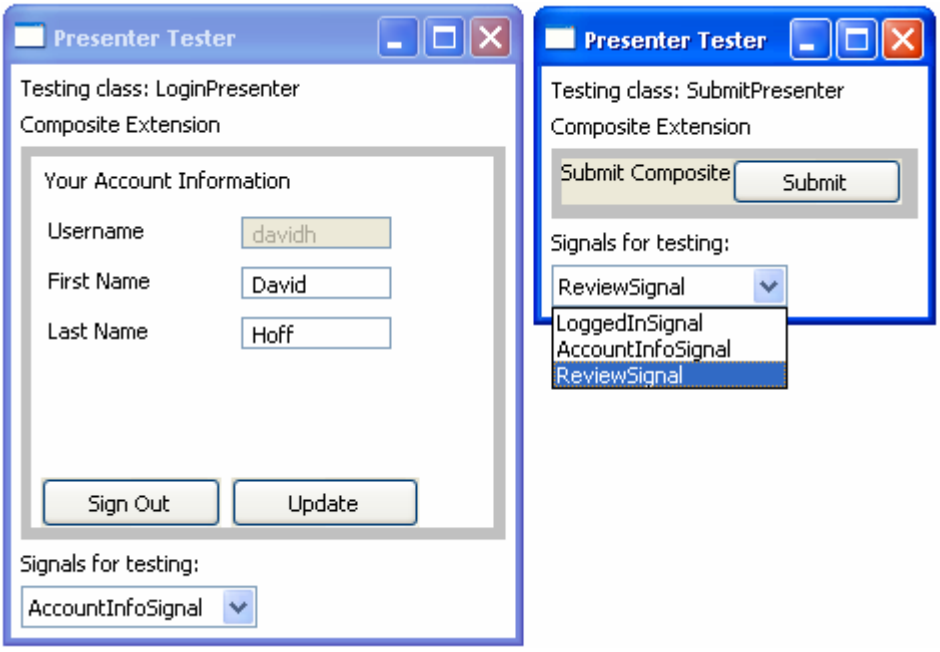

Figure 5-15: Presenter tester

Figure 5-15 shows the Login and the Submit presenters from the example application (5 - Re-implementation of the analysed application) in two separate testing environments. The view of the presenters is visualized in a composite structure extension.

The Login presenter is in LoggedIn state, Figure 5-10 above, after a UserInfoValid transition to CheckUser state and a ValidLogin transition to LoggedIn state. The first transition is triggered by a user event by pressing a button in the presenter. The second transition is triggered by a signal from the testing environment chosen in the dropdown box.

5.5 Comparison of the approaches

Here will we compare the analysed Case Management application, described in 3 - Analysis of a Case Management application, with the re-implementation using the middleware of this thesis. The re-implementation covers the problematic parts found in the analysed application, found in 3.3 - Problems with the application. The re-implementation is described in 5 - Re-implementation of the analysed application.

5.5.1 State management

The analysis of the CM application shows that the architecture offers no possibilities for state management of a module (See State management in 3.3). The state management has to be implemented in the module code itself. Different “states” has to be managed in all three parts of a module (Action, Form and JSP) and conditional logic has to be used to decide which “state” that is active.

The example re-implementation of the application has a fine grained level of state handling that the reference application lacks. The re-implementation uses UML state machines that execute signals asynchronously. UML state machines is visualised and gives an easy overview over the states of a module. The states are handled by the controller instead of three parts as in the CM application where states had to be managed. The controller used the view interface to set the state in the GUI.

One example is the controller of the registration presenter, Figure 5-8, which shows in an easy way the states of the module. The transitions show the behaviour the module implements. Figure 5-6 shows the look of the view in the different states. This shows the behaviour of the module with a precision that the CM application lacked.

This example illustrates that GUI design and implementation can have a high level of precision when it comes to describing the behaviour. It is easy to understand the transitions and see what the state machine does, and the implementation is guaranteed to have just one active state.

5.5.2 State management that changes other modules

If a module in the analysed application changes state, there is no standard functionality to inform other modules about the state transition. The only way of informing about a state change is to call that module directly.

The presentation middleware has the ability to be informed of state changes in other modules. A state transition can send a signal to other parts in the application. This can triggers state changes in the other parts.

Example from the re-implementation is the Submit presenter which has a controller that reacts to signals sent by other controllers. The controller reacts to signals sent by the registration controller and from the login controller (See Figure 5-9)

5.5.3 Navigation

As we have seen in section 3.3, where the problems with the analysed application where discussed, the architecture controls the navigation in a coarse grained fashion. Struts framework, described in 3.2.2, uses one configuration file for navigation. In the example implementation the navigation is controlled by state machines and asynchronous signals. The state machines are executed in parallel and no state machine can lock the others and freeze the application. This gives an improved control where all controllers are responsible for their own state management.

In the example with the search for users as seen in Figure 5-13, the controller of the Registration presenter, Figure 5-3, receives the user chosen from the search window while the window is still in control of the GUI. The name field in the Registration view, Figure 5-6, is updated in the background while the search window still got focus.

The middleware architecture gives a structure where the navigation can be modularised, and is easy to visualise. The simplicity of visualisation makes the communication concerning the intention of the application easier. It makes it easier to understand and grasp the extent of changes.

5.5.4 Long running transactions

In the analysed application, long running transactions stop the user in the middle of the step, in a process, until the step is completed. No asynchronous processing was introduced. In the presentation middleware all communication is asynchronous and no process is stopped unless it is desired. In the re-implementation, as described in 5 - Re-implementation of the analysed application, the user can continue with other tasks while waiting for results.

The asynchronous nature of the middleware makes management of long running transactions easy. A user task which triggers a transaction that will take long time to fulfil, runs in the background and the user will be able to continue with the work. After the completion of the transaction the user could be notified in an information view or the status bar etc.

An example from the re-implementation is the login of a user (Figure 5-11) where a delay has been added for illustration of a remote service response. The response is delayed for a couple of seconds, but the user can navigate to the registration view (the second view in Figure 5-6) and when the login is completed, the Submit button will be enabled. If the registration is in edit state, the user can continue to edit the registration while waiting for the login to be completed.

5.5.5 Notification

The analysed application had a simple notification which informed the users about administrative and operational events; see Notification in part 3.3 - Problems with the application.

The re-implementation has a notification implemented as a status bar text field which informs the user about changes in the application. In Figure 5-6 the status field is shown, and a transition in the Top Registration controller, Figure 5-5, changes the status triggered by a status signal.

A notification could be sent to all users at any time and the notification would be shown in the status bar when received. A notification history could easily be implemented using the log presenter, Figure 5-14, and by adding a filter to show only the signals of interest.

5.5.6 Misuse of the guidelines

The middleware has a strict constraint concerning where to put the logic for the different elements of the architecture. (See Table 4-1: Structure elements of the architecture)

- The view elements, the information in the view, can only be changed through the interface of the view. The view is only responsible for showing and changing the GUI.
- The controller has the logic for the state management and keeps the current state in the active state configuration. The controller uses the interface of the view to control the view.
- The presenter is only responsible for the connections to other presenters.

These constraints make the application modular and the parts developed are loosely coupled. The architecture forces a clear separation of concerns.

On the contrary, the analysed application needs logic in the presentation logic, the data carrier and in the presentation part of the application to perform a simple control of the view. (See State management in 3.3 - Problems with the application)

The clear separation of responsibility, which the analysed application lacks, is the strength of the presentation middleware.

5.5.7 GUI change

Logic for controlling the presence of GUI elements is placed in three parts in the analysed application (see GUI change in 3.3 - Problems with the application). In the re-implementation the controlling of the view is focused in the controller part of the presenter-controller- view triplet (4.3.3 - Controlling the interaction interface by state machines). This eases the maintainability of applications and removes common reasons for failures.

6. Conclusion and Further work

The thesis proposes an architecture for user interaction by using an asynchronous communication, execution of state machines and controlling of GUI elements in a hierarchical structure. This architecture has been implemented as a middleware that consists of a runtime environment, for the asynchronous communication and state machine executions, and base classes for patterns using the architecture.

6.1 Conclusion

The approach for user interaction which is shown in this thesis has many advantages compared to the common way of implementing user interaction, as seen in section 5.5, Comparison of the approaches.

6.1.1 Architecture

Standard UML state machines are used for controlling user interaction, 4.3.3 - Controlling the interaction interface by state machines. State machines are already used in the design instead of plain text which is usually the basis for design of user interaction. This makes the design more precise and many misunderstandings can be avoided. State machines can be visualized graphically. This makes the design easier to communicate to the stakeholders of an application.

State machines are a proven technique which has been used for different purposes in system engineering. State machines can also be verified by a model checking tools, performing formal verification by examining the set of possible executions of the model [del Mar Gallardo et al. '02].

The conclusions of the comparison of the analysed application and the re-implementation, 5.5.1 - State management, illustrate that GUI design and implementation can get a high level of precision by using state machines. It is easy to understand and examine the transitions to see what the state machine does. The implementation is guaranteed to have just one active state. The specification can be transformed to an implementation which in turn can be verified to the specification.

The approach used in Spring Web Flow, described in 2.2.3, has assets like the approach describe in this thesis. Sub flow navigation can be managed by a kind of composite state definition which can be reused. This is an improvement over the approach used in the Struts framework. But even if sub flow can be reused the definition is treated as a part of a global state machine and there is no execution of state machines in parallel or asynchronous messaging.

The asynchronous nature of the middleware

In SWT and other GUI libraries developers often abuse the Event Dispatch Thread; described in 2.4.2 - Standard Widget Toolkit.

The approach described in this thesis uses an asynchronous model. The model is close to the Actor language, described in 2.6 - Concurrency and asynchronous architectures, where the controllers have their own threads of control. Messages are sent to them asynchronous, queued and processed by the controllers own thread. The controller implements the designated behaviour of an Actor as transitions between states.

The parts modelled and implemented runs in a parallel environment without modification of the code or special modularization. (See 4.6.5 - The State Machine Runtime Engine) The environment in the middleware is, by design, thread ignorant. The developer does not have to own the thread model. The developer can not leave the thread problem completely, the parts have to be implemented in a thread safe manner as in a Java Enterprise application, but this is a considerable simplification of the implementation approach. This is in contrast to the standard way of implementing user interaction where the Event Dispatch Thread is executing the major part of the application and the developer has to know how to implement multithreaded applications.

Modularization

In the re-implementation of the analysed application we saw small parts of the application developed in isolation and tested. One example is the, from a GUI point of view, simple submit button which were a presenter on its own and hooked into the Registration presenter (Figure 5-6: Edit and reviewing the registration).

With the built-in modularization of the architecture where the parts are connected only by the signal, API testing of the modules is easy. Common used test frameworks like JUnit, with a small extension, can be used to test the behaviour of modules independent of other modules. There is also a testing environment where modules can run separate using the GUI for interaction, 5.4 - Testing the presenters. Here the presenters and their belonging controller and view are tested in isolation. Signals can be registered into the testing environment and submitted to the tested presenter and the result is displayed in the testing GUI.

Extendability

When using this architecture it is relatively simple to extend an application. Presenters can offers extension points where children can add their GUI and behaviour. An example is the stopwatch application with the tab folder extension. Children can add tabs with GUI and functionality as seen in Figure 4-20: The stopwatch view. The stopwatch starts with an empty tab folder and is extended with a results view. The look of a view can be changed by adding a new view implementation with the same interface the controller uses. The controller and the presenter are reused. (See Figure 4-8: Controller - View structure)

6.1.2 Features of the middleware

The middleware gives a great possibility for personalization of applications. Due to the dynamic behaviour of the architecture and the loose coupling between the parts involved in the application it gives an easy way to separate behaviour for different users. The functionality of an application could change to be more specialized due to the need of the user. Views or parts of views could be changed or added to an existing application, at runtime. An application could start simple with the basic functionality which covers the need for the most users. When the user has more demanding needs additional behaviour is added on demand. One example is the dynamic adding of result view and additional stopwatches in 4.4.4 - Dynamic behaviour.

Covers the need of the blur of applications

In the discussion about the blur of applications, section 2.1 - Development in Human Computer Interaction, the need for dynamically altering of applications was stated. The architecture described in this thesis has abilities to both extend and remove functionality runtime. This ability makes this approach an ideal fit for applications that need to be able to be extendable for different purpose. The runtime extension, described in 4.6.4 - Dynamic behaviour, adds presenters to the application by using the same general functionality that is used otherwise in the architecture; the new presenter is passed along with a signal to a controller.

The architecture also makes it easy to let an application span across computers since the communication between different virtual machines is contained in the architecture. The example stopwatch application, Figure 4-19, has the top presenter in a one JVM and the children in separate JVM's. The presenters of the application can run in any device which can execute a JVM.

6.2 Further work

The implementation and the analysis of the middleware should be investigated further. There are many ideas that should be tried out in a realistic application and used by real users to see how the middleware works. This would gather useful experience with the modelling of applications using the middleware.

6.2.1 Architecture of the Middleware

Various aspects of the architecture could be experimented with.

This thesis uses a presenter hierarchy as the basis for the middleware. While gathering experience from applications using the middleware other organization could be investigated. The simplicity of the presenter, view, controller structure is efficient but there might be

needs that could benefit from using other structures. For examples the broadcast bounce could pass the signals to the top presenter without executing the signals and just perform the execution of the signals on the way down for efficiency.

No performance test of the middleware has been conducted during the work with the thesis. This has to be performed to figure out how the architecture performs and scales in a larger environment spanning many users and JVM's.

When an application is closed the state of the controllers is lost. There is no persistence of the state machines built in to the middleware. It should be possible to store the state machines and presenter structure since they are ordinary properties in Java classes.

Traceability

Debug and tracking of the execution of events and state changes needs to be achieved in another way than usually performed in single thread applications. The usually call stack method, where a thread is traced through the execution, is insufficient since the execution of the controllers happens in parallel.

There is a need for tracing state changes accomplished by some kind of listener. This listener could be another application that is informed over a socket connection. Another idea is to use a standard part of the middleware for tracing. The architecture can submit a specialized signal, in event of a state change, which can be logged by a presenter.

Reliability

There are no ACID transactions (Atomicity, Consistency, Isolation, and Durability) in the architecture. The architecture has no guarantee that a signal from a presenter will be returned in response to a request. The reliability has to be in the communication protocol which is implemented in the application. If a part of the application need a response to a signal sent, a timeout transition can be added to the controller. If no response is received in a predetermined time, a new signal could be sent with the same request.

This is an area that needs further investigation. Which communication interactions are the most effective and easiest to maintain?

Security

There is no form of security implemented in the middleware. Due to the openness of the middleware, security and authorization is important to protect applications from unwanted behaviour. Who should be allowed to send signals that could create presenters? Is the information in the signal of interest to every presenter in the hierarchy or should some information be protected in some way? The API of a presenter could include some kind of security information of how to use the information in the signals. This API information should just be shared with trusted developers.

Code generation

There have been some experiments with code generation from UML models to be used with the middleware runtime (4.6.6 - Model Driven Architecture). To shorten the development cycles a code generation tool would be useful. The construction of a code generator should be achievable due to the strict structure of the middleware architecture. When a code generator is constructed for one GUI library it should be straightforward to adapt the generator to other libraries.

6.2.2 Application experiments to investigate

To gain experience with the usage of the middleware and modelling for the architecture experiment applications must be implemented. The experiments should have extended use of the dynamic behaviour and structures for communication where presenters are in remote locations. There should also be experiments with presenters that come and go like in an ad hoc network and how the architecture could cope with this kind of situations.

One interesting experiment could be an application that spans different environments in one single application. The application could range from a desktop application to mobiles and sensors. One example is an application that monitors patients. Sensors added to the patients submit signals in the application hierarchy, and the desktop part of the application monitors and visualize the signals received. If a signal received by a sensor, has a value below some limit, a warning signal could be sent both the desktop application and the mobile phone.

Reference:

Acceleo Acceleo. Obeo.http://www.acceleo.org/

Alexander, C., Ishikawa, S. & Silverstein, M. (1977) *A pattern language: towns, buildings, construction,* New York, Oxford University Press.

Alur, D., Malks, D. & Crupi, J. (2003) *Core J2EE patterns : best practices and design strategies,* Upper Saddle River, NJ, Prentice Hall.

Anderson, D. J. (2000) Using MVC Pattern in Web Interactions.http://www.uidesign.net/Articles/Papers/UsingMVCPatterninWebInter.html

Anderson, D. J. O. B., Brían (2003) Lean Interaction Design and Implementation: Using Statecharts with Feature. *ForUse 2003.*

Anderson, D. O. C., Marcus; Byrne, Martin; Mcsherry, Mark (2007) JStateMachine.http://jstatemachine.sourceforge.net/

Avgeriou, P. & Zdun, U. (2005) Architectural patterns revisited–a pattern language. *Proceedings of 10th European Conference on Pattern Languages of Programs.*

Beers, G. A. & Carey, J. (2006) WebSphere Process Server Business State Machines concepts and capabilities. IBM.http://www-128.ibm.com/developerworks/websphere/library/techarticles/0610_beers/0610_beers.html

Booch, G., Jacobson, I. & Rumbaugh, J. (2005) *The unified modeling language user guide,* Upper Saddle River, N.J., Addison-Wesley.

Borland (2007) Borland Together.http://www.borland.com/together

Bræk, R. (2000) On Methodology Using the ITU-T Languages and UML. TELEDIREKTORATET

Budinsky, F., Steinberg, D., Raymond Ellersick, R., Merks, E. & Grose, T. (2003) *Eclipse Modeling Framework*, Addison Wesley.

Börger, E. & Stärk, R. (2003) *Abstract state machines: a method for high-level system design and analysis,* Berlin, Springer.

Carnell, J. & Harrop, R. (2004) *Pro Jakarta Struts*, Apress.

Cavaness, C. (2002) *Programming Jakarta Struts*, O'Reilly.

Cocoa (2007) Cocoa application environment for Mac OS X. Apple.http://developer.apple.com/cocoa/

Coutaz, J. (1987) PAC, an Object Oriented Model for Dialog Design.

Del Mar Gallardo, M., Merino, P. & Pimentel, E. (2002) Debugging UML designs with model checking.

Eckstein, R. (2007) Java SE Application Design With MVC. *Sun Developer Network.*http://java.sun.com/developer/technicalArticles/javase/mvc/

Feathers, M. (2002) The Humble Dialog Box. Object Mentor, Inc.www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf.

Firefox Firefox Web browser. Mozilla Foundation.http://www.mozilla.org/firefox/

Fowler, M. (2003) *Patterns of enterprise application architecture,* Boston, Addison-Wesley.

Fowler, M. (2004) Inversion of Control Containers and the Dependency Injection pattern.http://www.martinfowler.com/articles/injection.html

Frankel, D. S. (2003) *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley.

Gamma, E. (1994) *Design patterns : elements of reusable object-oriented software,* Reading, Mass., Addison-Wesley.

Gamma, E. & Beck, K. (2004) *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*, Addison-Wesley Professional.

Gartner (2006) 10 Year Scenario for Information Technology, Business and Society. IN PRENTICE, S. (Ed.), Gartner

Gurevich, Y. (2004) Abstract state machines: An overview of the project. Springer

Harel, D. (1987) Statecharts: A visual formalism for complex systems.

Haugen, Ø. & Møller-Pedersen, B. (2000) JavaFrame: Framework for Java-enabled modelling. *ECSE2000.* Stockholm,

Hewitt, C. (1976) Viewing Control Structures as Patterns of Passing Messages. AIM-410

Hibernate (2007) Hibernate Relational Persistence. JBoss.http://hibernate.org

Hohpe, G. (2006) Programming Without a Call Stack–Event-driven Architectures. *ObjektSpektrum.*

Husa, K. E. & Melby, G. (2004) ActorFrame Architectural Guide. NorARC

Hyman, P. (2005) The Watches That Watch You. *The Wall Street Journal.*

Johnson, R. (2002) *Expert one-on-one: J2EE design and development,* Birmingham, WROX Press.

Johnson, R. & Hoeller, J. (2004) *Expert one-on-one J2EE development without EJB*, Wiley.

Kiczales, G. (1996) Aspect-oriented programming. ACM Press New York, NY, USA

Mainframe, I. (2007) IBM Mainframe web.http://www.ibm.com/systems/z/

Markopoulos, P. (1997) *A Compositional Model for the Formal Specification of User Interface Software*, University of London.

Martin, R. C. (2003) *Agile software development : principles, patterns, and practices,* Upper Saddle River, N.J., Pearson Education.

Mcclanahan, C. R. (2007) Struts. Apache Software Foundation.http://struts.apache.org/

Melby, G. (2003) Using J2EE Technologies for Implementation of ActorFrame Based UML2. 0 Models. Grimstad, Master Thesis in Information and Communication Technology, Agder University College, Grimstad, May 2003

Mitchell, J. C. (2003) *Concepts in Programming Languages*, Cambridge University Press.

Mq, I. (2007) IBM MQ.http://www.ibm.com/software/integration/wmq/

Nintendowii Wii. Nintendo.http://wii.com/

Noakes, M. D., Wallach, D. A. & Dally, W. J. (1993) The J-machine Multicomputer: An Architectural Evaluation.

Oaw openArchitectureWare.http://www.openarchitectureware.org/

Omg (2007) The Object Management Group.http://www.omg.org/

Omg.Org/Uml (2007) Unified Modeling Language Resource Page. Object Management Group.http://www.uml.org/

Petzold, C. (2006) *Applications= code+ markup: a guide to the Microsoft Windows presentation foundation*, Microsoft Press.

Raistrick, C. (2004) *Model driven architecture with executable UML*, Cambridge University Press New York.

Reenskaug, T. (1979a) MODELS-VIEWS-CONTROLLERS.

Reenskaug, T. (1979b) THING-MODEL-VIEW-EDITOR-an Example from a planningsystem.

Reenskaug, T. (2003) The Model-View-Controller (MVC) Its Past and Present.

Rumbaugh, J., Booch, G. & Jacobson, I. (2004) *The unified modeling language reference manual,* Boston, Addison-Wesley.

Seitz, C. L. (1985) The cosmic cube. ACM Press New York, NY, USA

Sells, C. & Griffiths, I. (2005) *Programming Windows Presentation Foundation*, O'Reilly.

Simon, H. A. (1996) *The Sciences of the Artificial*, MIT Press.

Singh, I., Stearns, B. & Johnson, M. (2002) *Designing Enterprise Applications with the J2EE Platform*, Prentice Hall PTR.

Springframework (2007) Springframework.org. Interface21.http://www.springframework.org

Statesoft (2007) Statesoft.com.http://www.statesoft.com/

Swithinbank, P. (2005) *Patterns Model-driven Development Using IBM Rational Software Architect*, IBM, International Technical Support Organization.

Swt, E. (2007) The Standard Widget Toolkit. The Eclipse Foundation.http://www.eclipse.org/swt/

Trolltech (2007) Trolltech.http://www.trolltech.com/

Walrath, K. (2004) *The JFC Swing tutorial: a guide to constructing GUIs,* Boston, Mass., Addison-Wesley.

Willersrud, A. (2006) User-defined code generation from UML 2.0. Oslo, A. Willersrud

Wps (2007) WebSphere Process Server. IBM.http://www-306.ibm.com/software/integration/wps/

Appendix A - Containment in the enclosed disc

The disc contains the implemented architecture described in the thesis and example applications. The architecture and the examples are implemented in Java by using Eclipse as the IDE. Java 5 is used for the development. To import into Eclipse choose ‘Import’ and ‘Existing Projects into Workspace’.

Structure of projects

StateMachine - This contains the runtime environment and the basic classes for the state machines as action, guard, transition etc.

StateView - The StateView contains the view library built upon the StateMachine engine runtime. It is presenters, controller, view, specialized actions, testing environment and utilities classes for creating presenters etc.

StateViewExample - The example project contains the calculator and the stopwatch example.

ExampleApps - The re-implementation described in 5 - Re-implementation of the analysed application is implemented in this project.

Instructions

The easiest way or running the examples is to use the Eclipse IDE where the SWT jar is included.

Calculator example

Run TopCalculatorPresenter as SWT application.

Stopwatch example

Run TopStopWatchPresenter as SWT application.

Stopwatch example with a central presenter

Run StopWatchCentral as Java program with the following program arguments: listenport=4444

Run TopStopWatchPresenter as SWT application with the following program arguments: parenthost=127.0.0.1 parentport=4444

The Re-implementation example

Run CentralRegistrationPresenter as Java program with the following program arguments: listenport=4443

Run TopRegistrationPresenter as SWT application with the following program arguments: parenthost=127.0.0.1 parentport=4443